

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

José Rafael Vasconcelos Cavalcante

January 5, 2011

1 Introdução

2 Configuração de ambiente de desenvolvimento

Neste capítulo será 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 será 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* , será o *Eclipse Luna* . O *Gradle* será usado como *build tool* e o *Apache Tomcat* como *container Java Enterprise Edition* (*JEE*). Para desenvolver em *C#* , será 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.

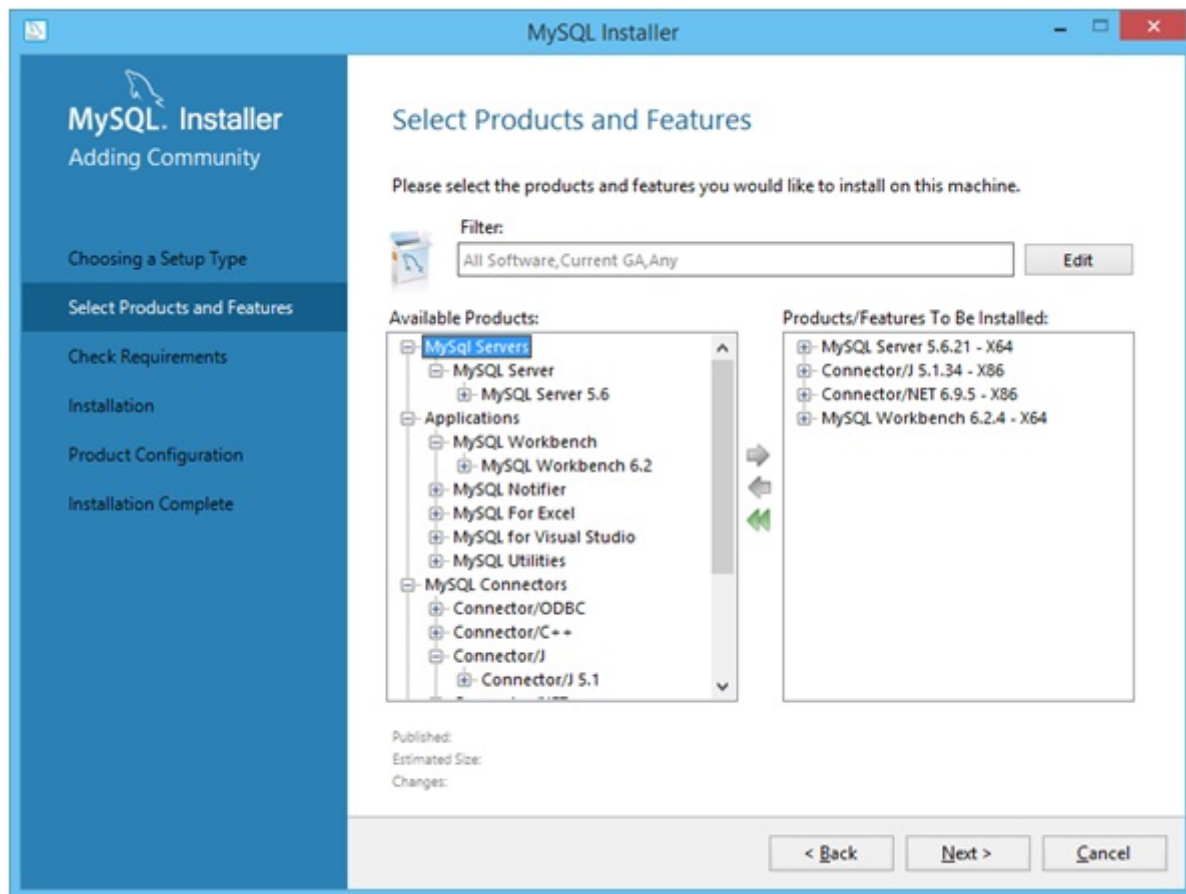


Figura 1: Opções de instalação do *MySQL*

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, será 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 instância do MySQL.

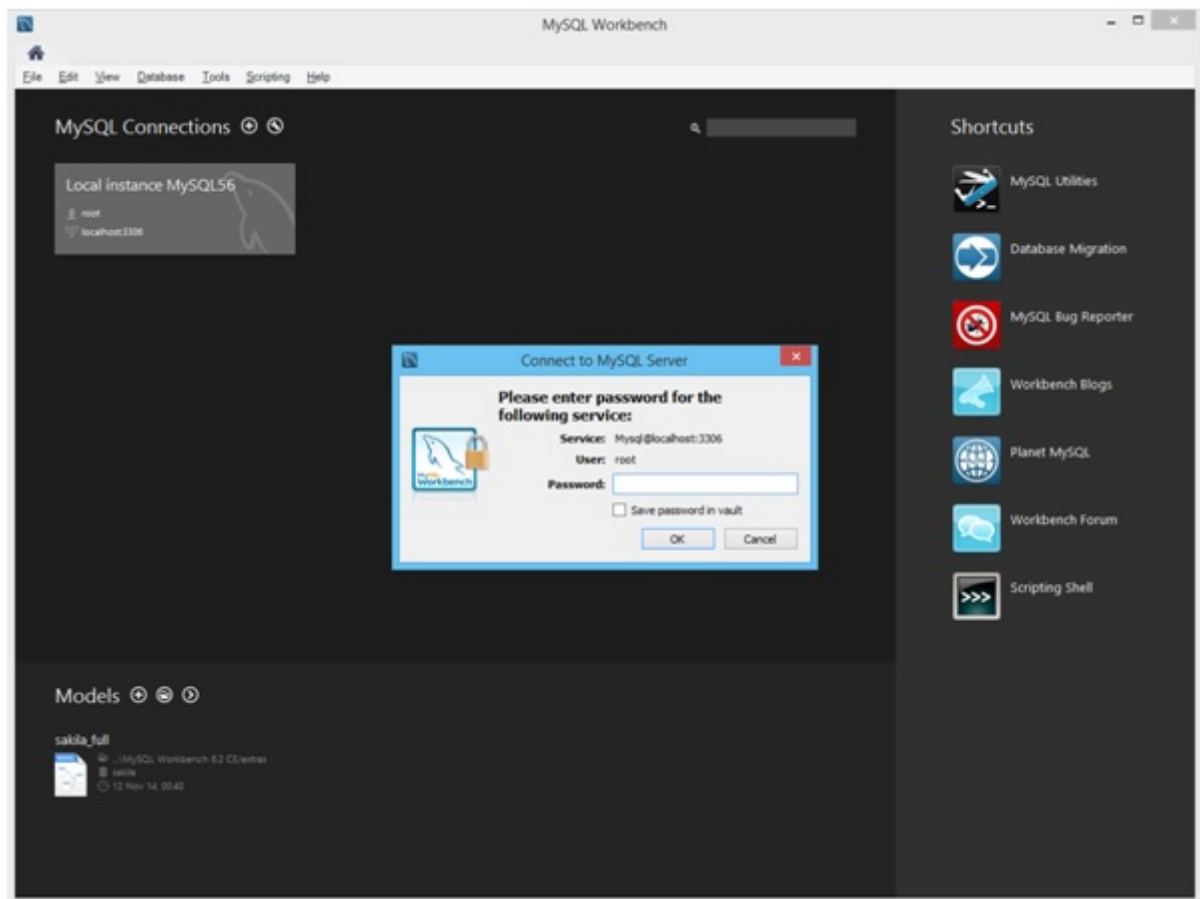


Figura 2: O MySQL Workbench

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, será utilizado o Eclipse Luna e o Java Development Kit 8 (JDK 8). Será usado o Gradle como build tool através de um plugin do Eclipse e o servidor web utilizado será 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, será usado nesse trabalho a versão para Windows de 64 bits.

Para fazer a instalação do JDK, foi 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 será mantido o diretório padrão como mostrado na figura 3.

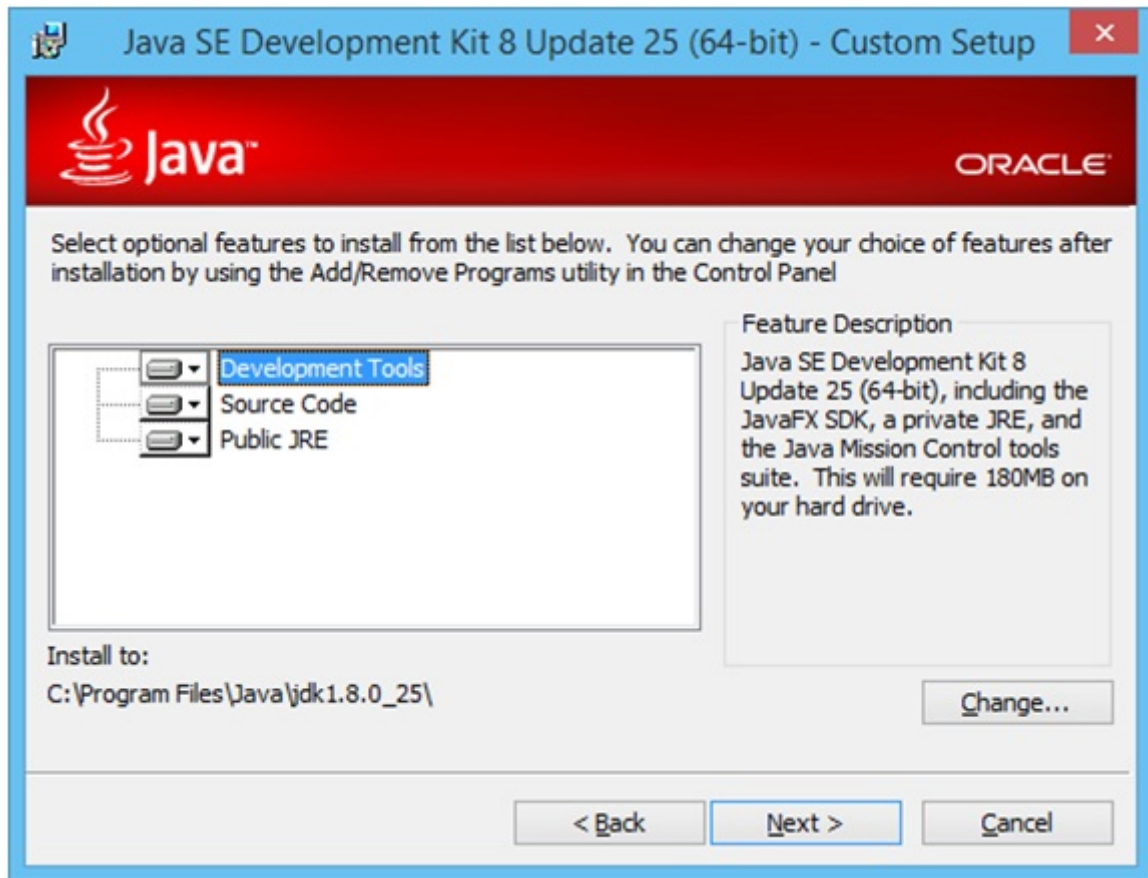


Figura 3: O instalador do JDK 8

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.

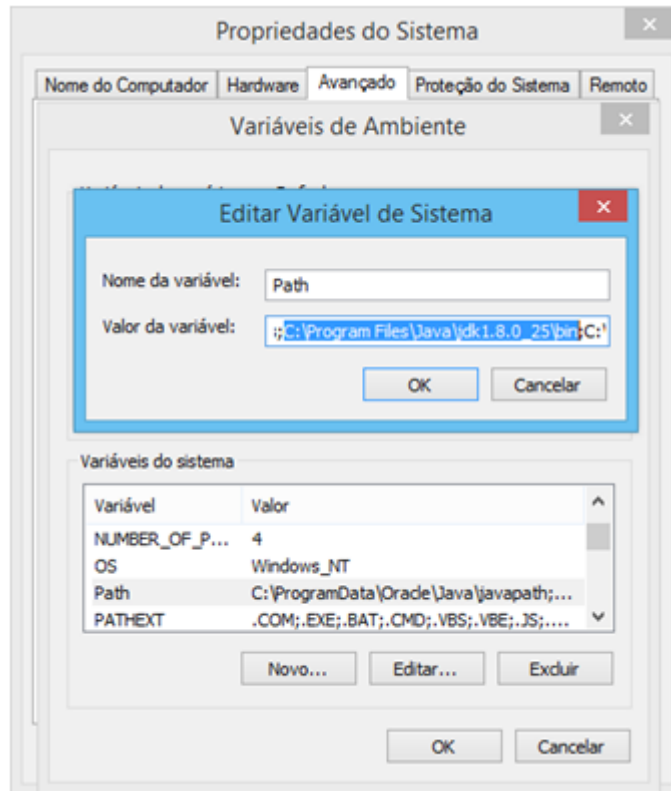


Figura 4: Configurando a variável PATH

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 existir problemas, será 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 será exibida uma janela para configurar o Workspace padrão (uma pasta onde serão guardados projetos e configurações), como está ilustrado na figura 5.

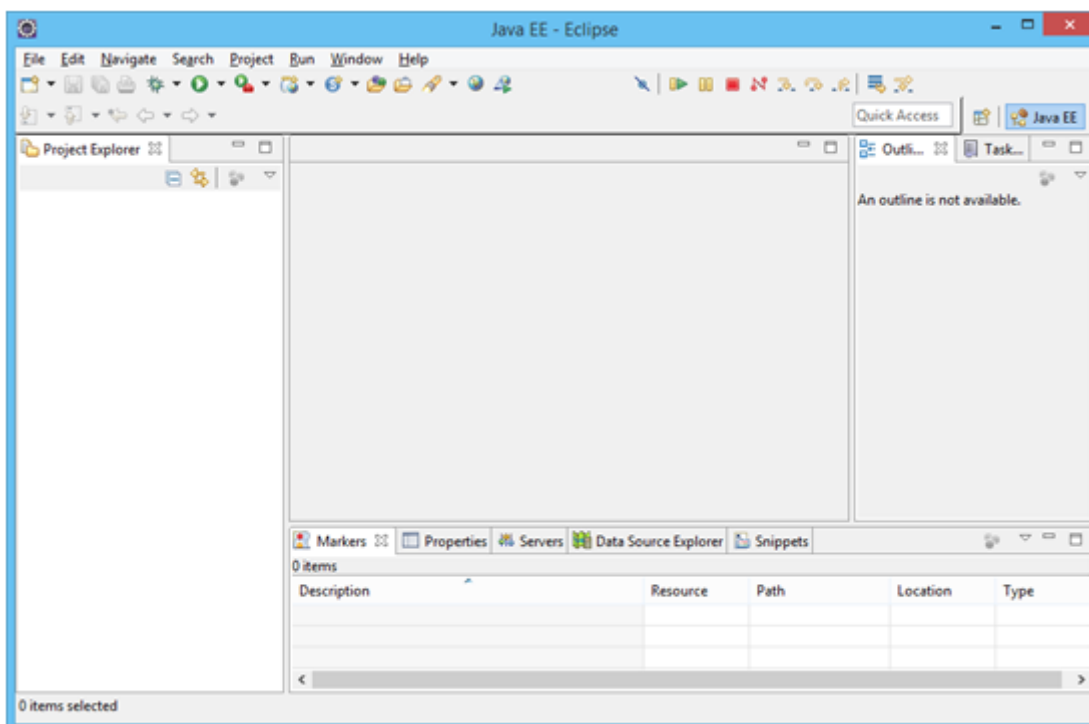


Figura 5: Eclipse recém instalado

2.2.3 Instalando o plugin do Gradle

O Gradle é a build tool que será 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 contínua), mas será 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 será 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 árvore de tipos de projetos, no menu de novos projetos, como pode ser observado na figura 6.

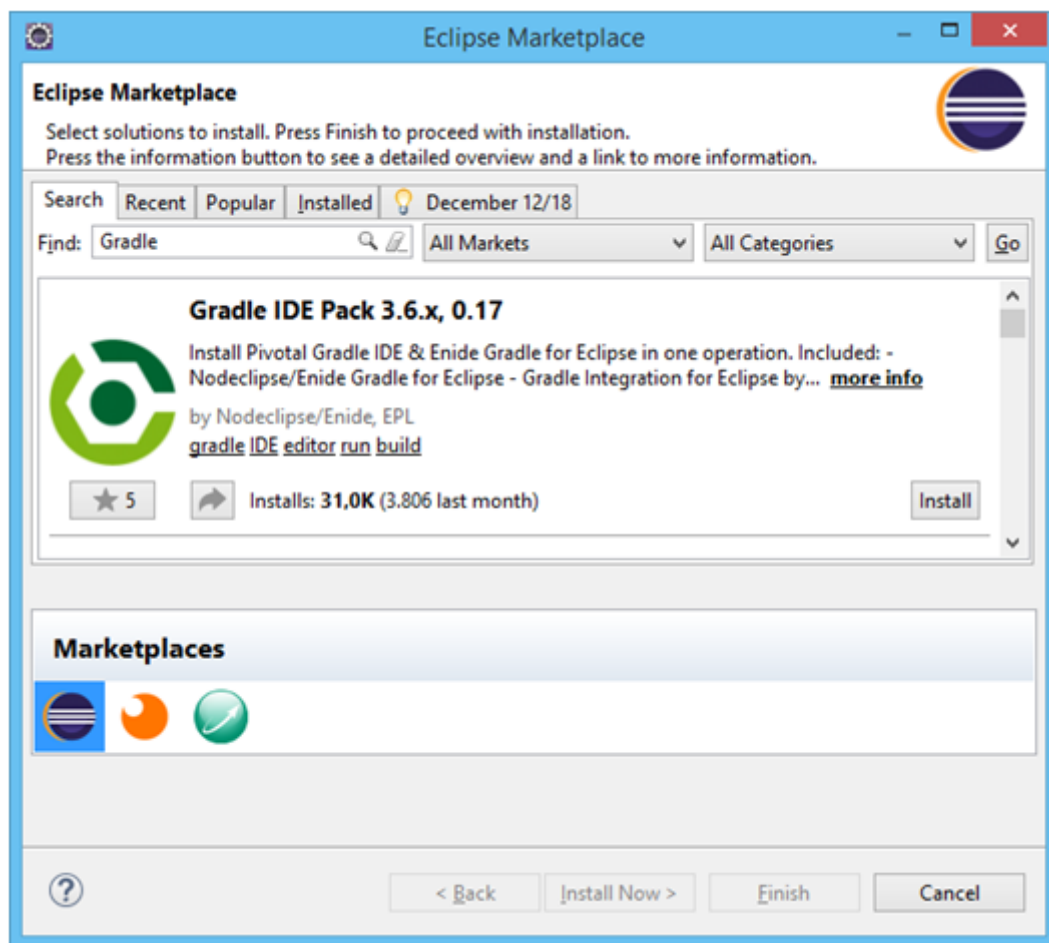


Figura 6: Instalando o plugin do Gradle

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 será 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 será usada a raiz do disco C:).

Na aba "Servers" do Eclipse possui 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 no nosso exemplo). Clicando no botão

“Finish”, o Tomcat está pronto para uso com o Eclipse.

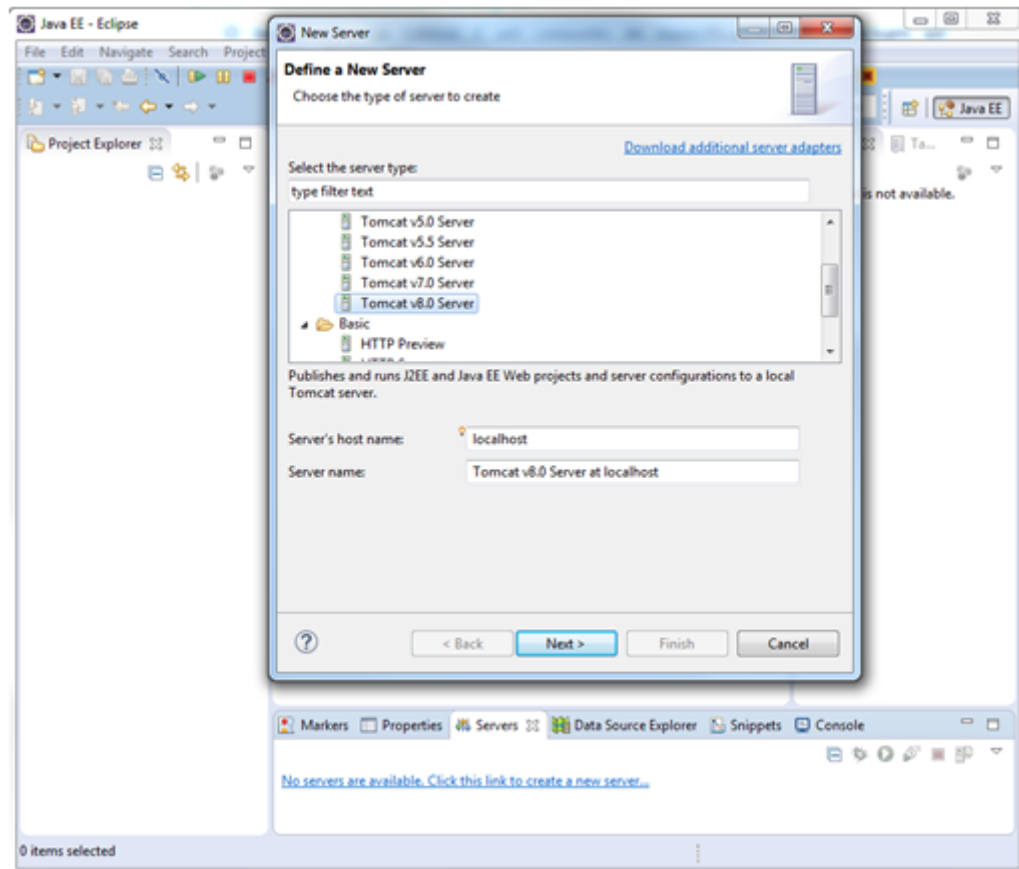


Figura 7: Janela pra adicionar servidores no Eclipse

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 irá baixar 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>.

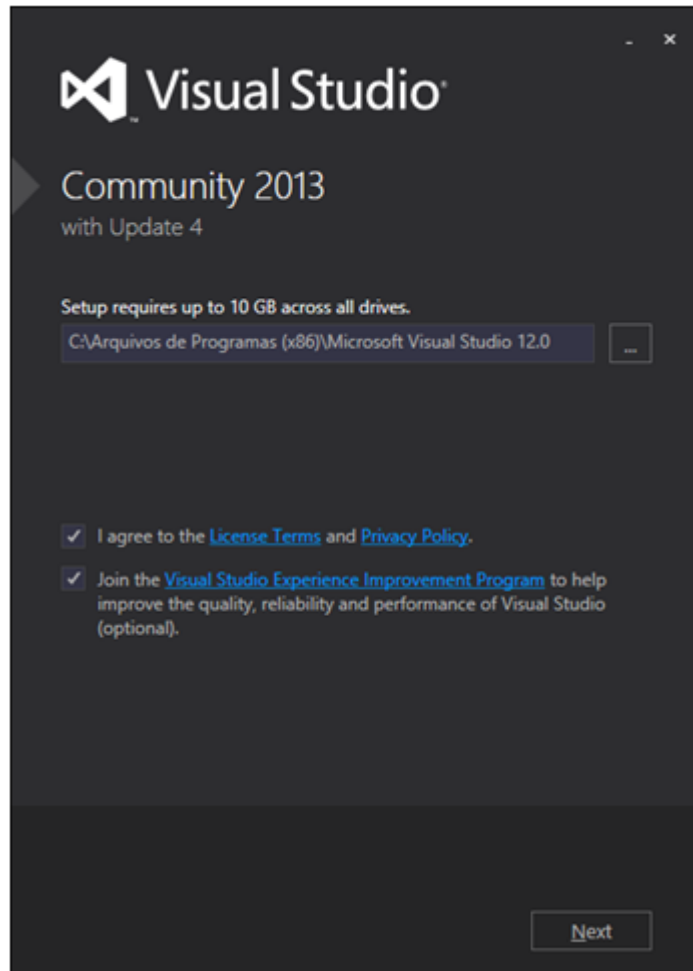


Figura 8: Instalador do Visual Studio Community 2013

Clicando no botão “Next”, a próxima tela que o instalador irá exibir 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”.

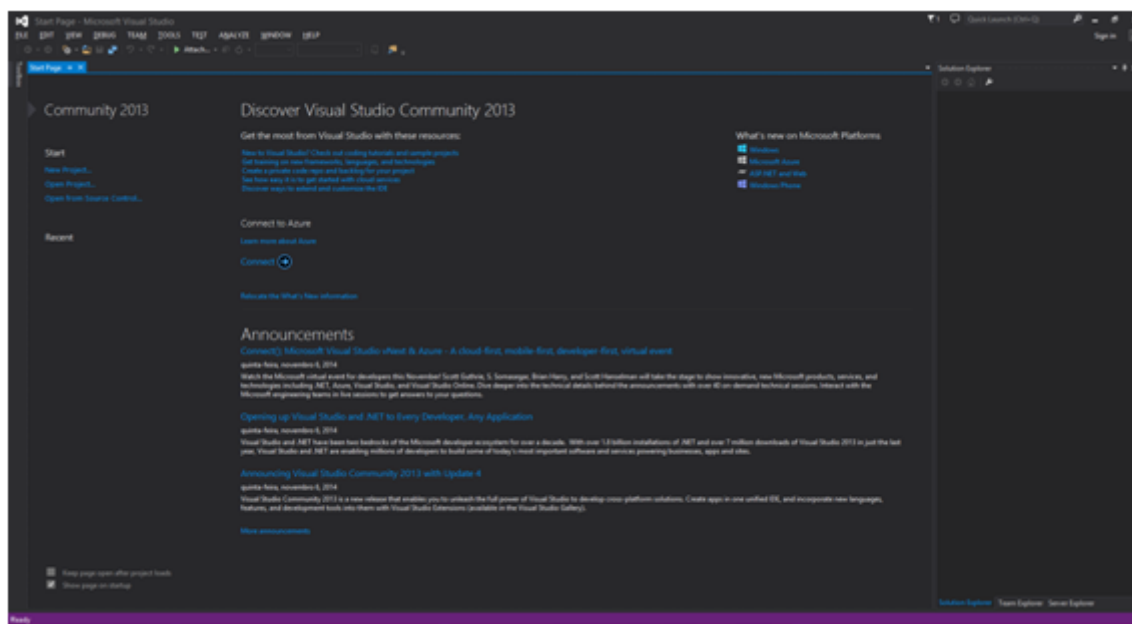


Figura 9: Tela inicial do Visual Studio Community 2013

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.

As vantagens 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 é a 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 desne-

cessários e tomando espaço em disco.

No próximo capítulo será demonstrado como criar um projeto de uma aplicação web nas duas plataformas.

3 Criando projetos web no padrão MVC

Nesse capítulo será 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 será 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 será o Gradle Project. Esse modelo de projeto está localizado na pasta Gradle na árvore de novos projetos do Eclipse. Na figura 10 pode-se observar a localização do modelo e a criação do novo projeto.

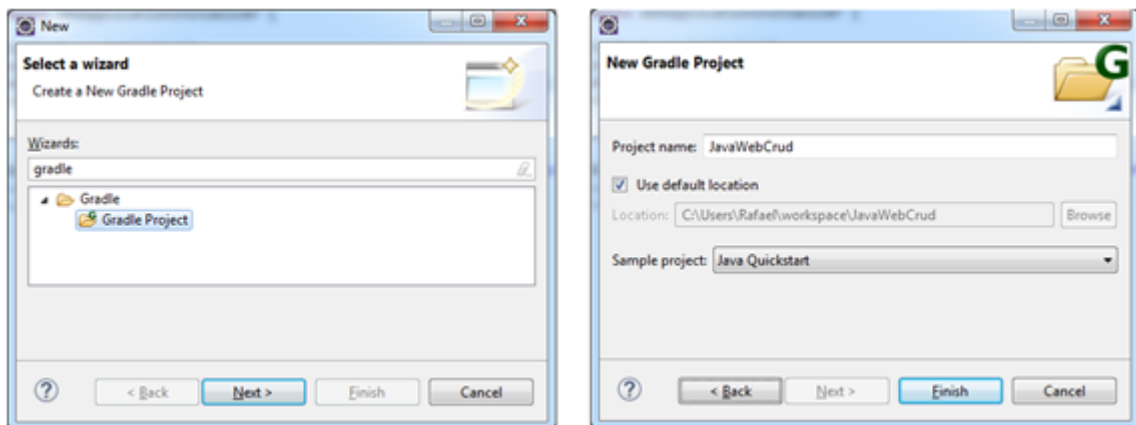


Figura 10: Criando um projeto do Gradle

Será criado um projeto com a estrutura mostrada na figura 11.

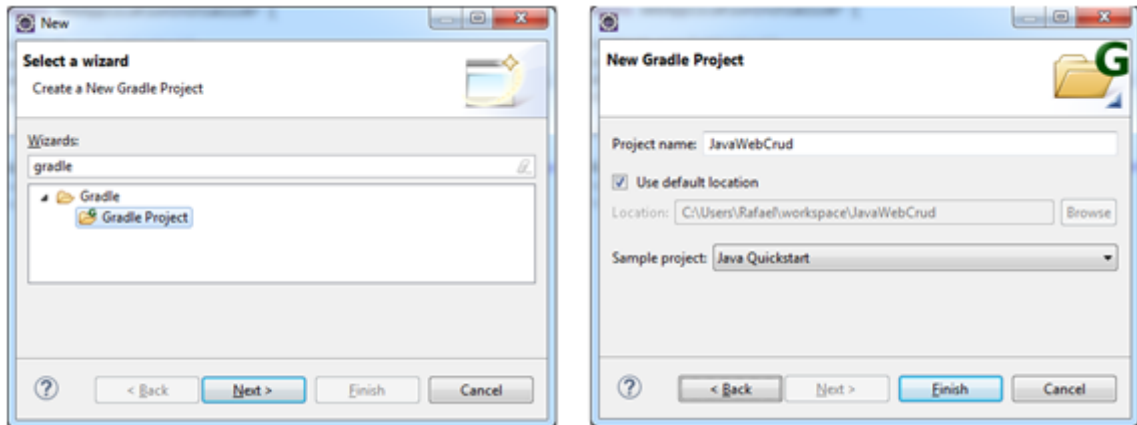


Figura 11: Estrutura inicial de um projeto do Gradle

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 nossos exemplos e serão 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'
        }
    }
}
```

Quadro 1: O arquivo build.gradle

A estrutura do projeto e o arquivo gradle.build gerados devem ser modificados para servir à uma aplicação web. Será adicionada uma nova pasta, chamada de WebContent, para armazenar conteúdo específico 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.

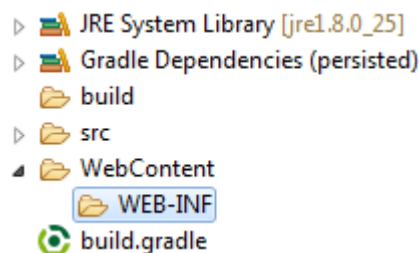


Figura 12: Estrutura do projeto com a pasta WebContent

Em seguida, é necessário modificar o arquivo gradle.build. 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 específico para web, será 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, será utilizado o plugin eclipse-wtp. O início do arquivo gradle.build deverá ficar como no exemplo do quadro ??.

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

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

Quadro 2: Arquivo gradle.build 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 (tasks) 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. gradle-command.png

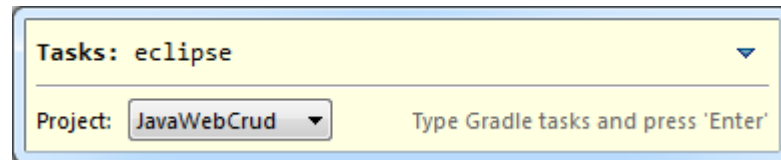


Figura 13: Gradle Task Quick Launcher

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/<caminhodaaplicaç~ao>`. Esse endereço será 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, será 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 será demonstrado como configurar o servlet padrão e o Spring MVC, no capítulo 3 será demonstrado como se criam controllers e views. tomcatproject.png

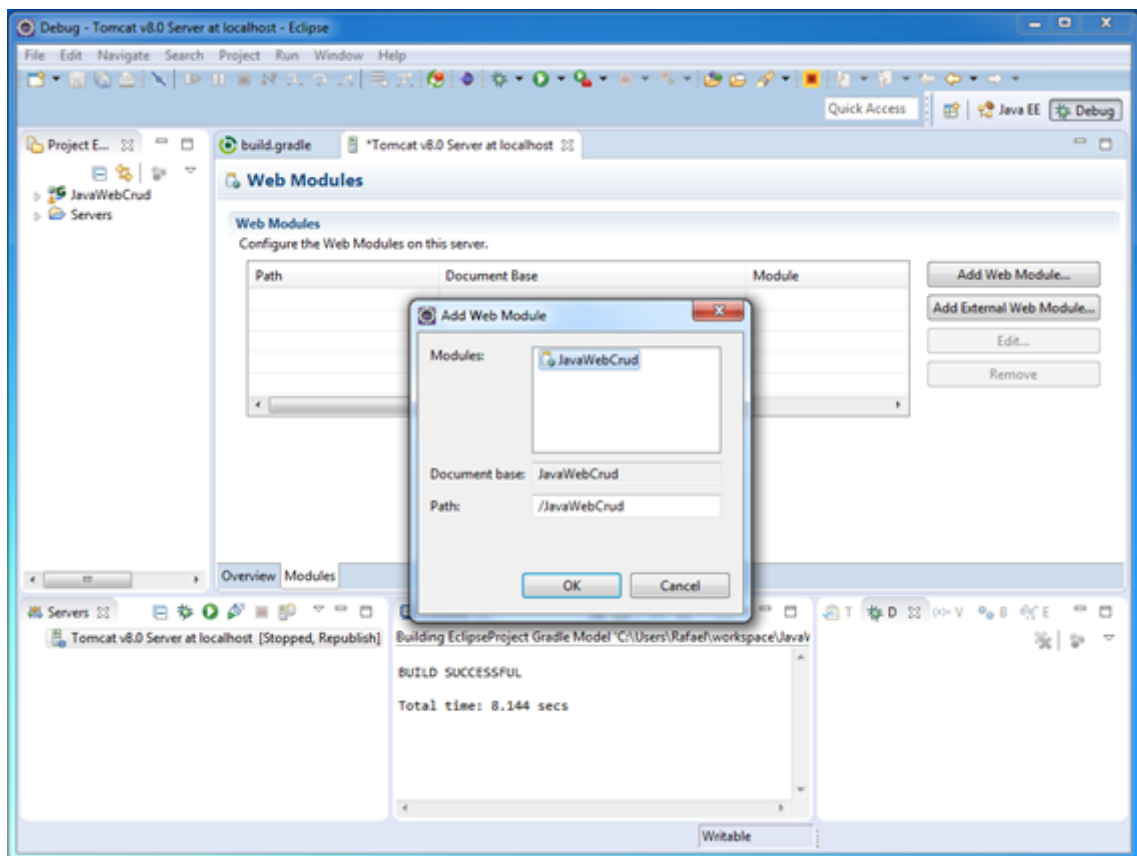


Figura 14: Adicionando um projeto web ao Tomcat

3.1.2 Adicionando dependências ao projeto

Abaixo estão listadas as bibliotecas que serão usadas ao 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 serializa 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 serão 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'
}
```

Quadro 3: Adicionando dependências ao projeto

Com o arquivo gradle.build alterado e salvo, o comando para baixar as dependências para o projeto estará localizado no menu de contexto do projeto (clcando 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.

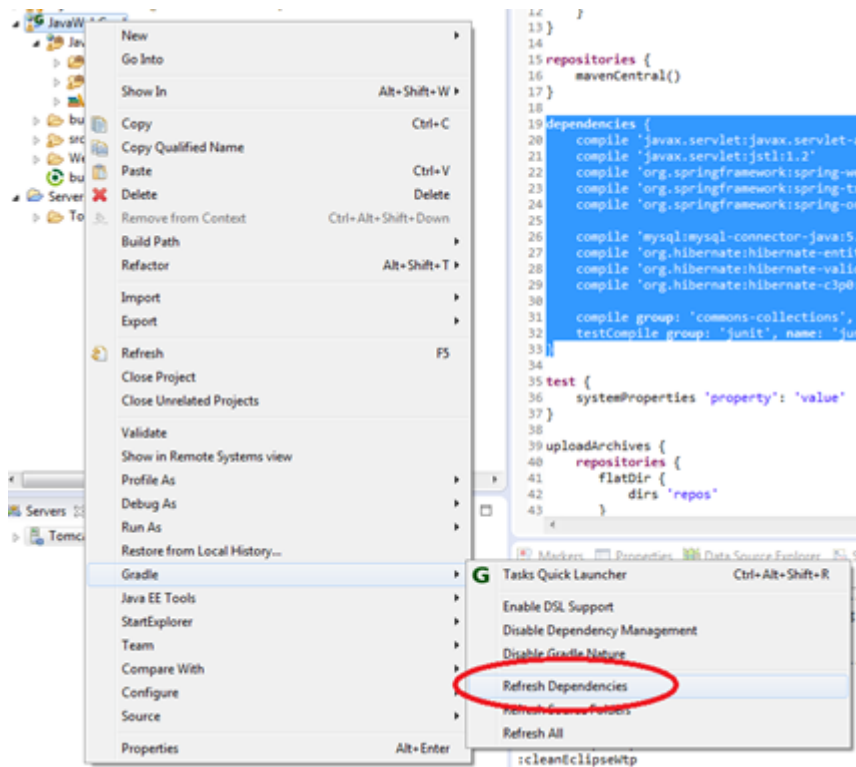


Figura 15: Atualizando dependências de um projeto Gradle

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, será usada a segunda opção.

Os arquivos de configuração são armazenados em um pacote chamado `br.uece.webCrud.conf`. Dentro desse pacote são criadas duas classes, `AppInitializer` e `SpringMvcConfig`, como mostra a figura 16.

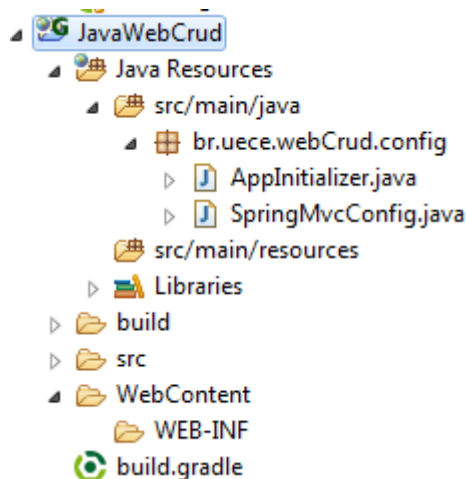


Figura 16: Estrutura do projeto *Java / Spring MVC* com pacote para arquivos de configuração

A primeira configuração que deve ser feita é usar a classe `DispatcherServlet` como o servlet padrão e configurar o contexto da aplicação (configurações específicas 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 herdaram 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 a 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` será 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 será usado para todas a páginas web e demais caminhos dentro da aplicação. O método `setLoadOnStartup` recebe como parâmetro o valor 1, para configurar nosso 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(
            springServletRegistration.addMapping("/");
            springServletRegistration.setLoadOnStartup(1);
        }
}

```

Quadro 4: Classe AppInitializer

Vários objetos que fazem parte da estrutura da aplicação (também conhecidos como Spring Beans) serão configurados na classe SpringMvcConfig, como mostra o quadro ???. 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"})
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 configurator) {
        configurator.enable();
    }

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurator) {
        {
            configurator.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 contentNegotiatingViewResolver()
    {
        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;
    }
}

```

Quadro 5: Classe SpringMvcConfig

A classe começa definindo várias propriedades que serão usadas para conexão como 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 (nor-

malmente 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 Servlet Handler padrão do Spring e os 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 foi 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 `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 serão 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 Apress.

O último método retorna um objeto `JpaTransactionManager`. Esse objeto será o responsável por gerenciar as transações com o banco de dados. Ele recebe a instância do objeto `LocalContainerEntityManagerFactoryBean` configurado anteriormente como parâmetro.

3.2 Criando um projeto *ASP.NET MVC 5* no Visual Studio 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, será 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 no Visual Studio aos arquivos de projeto do Eclipse e o arquivo gradle.build. vsnewproject.png

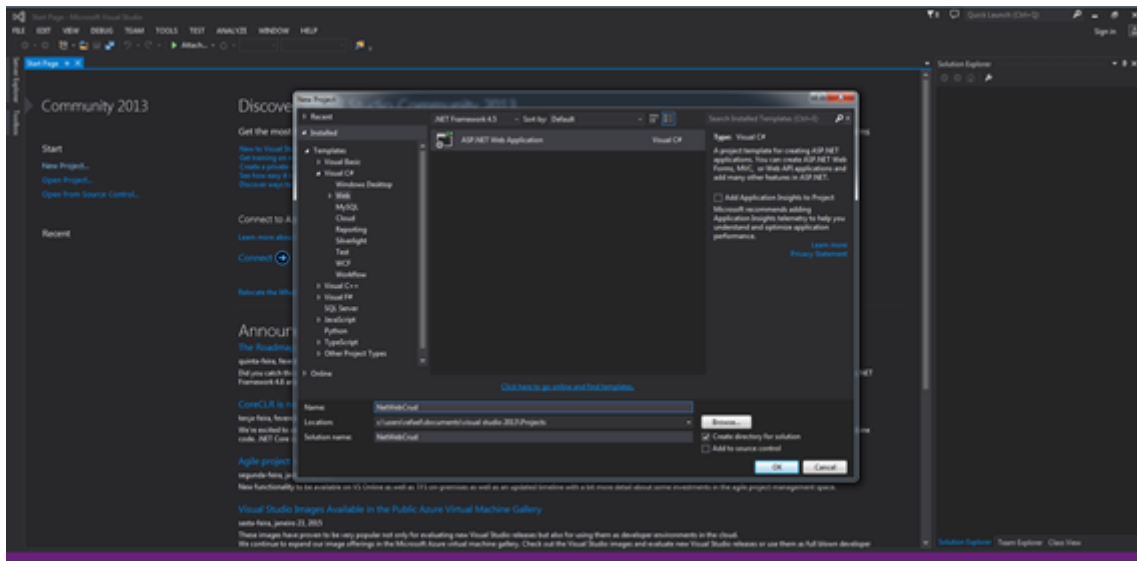


Figura 17: Criando um novo projeto no Visual Studio Community 2013

O nome da solução de exemplo será “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.

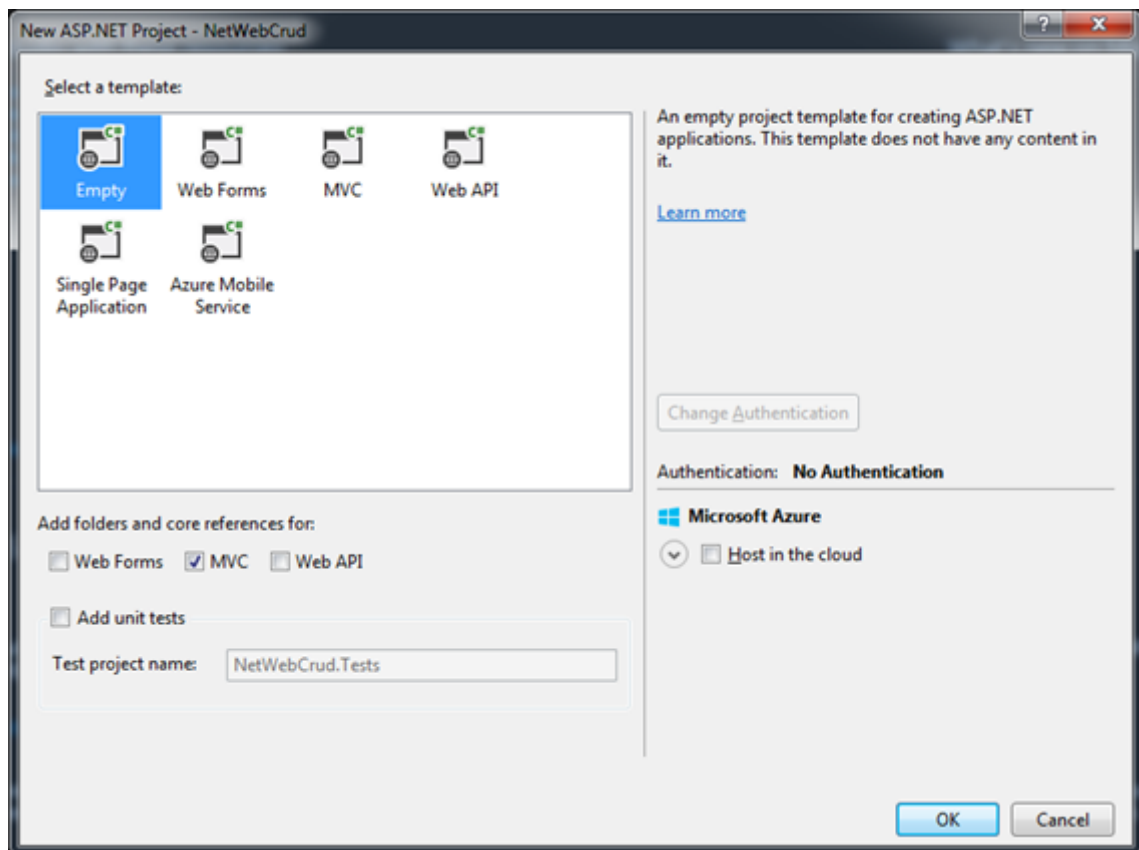


Figura 18: Opções de um novo projeto web no Visual Studio

Existem várias opções de projetos para web. Para um projeto MVC contendo apenas o mínimo necessário para seu funcionamento, será escolhido o modelo “Empty” e é marcada a opção MVC. O serviço Microsoft Azure não será 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.

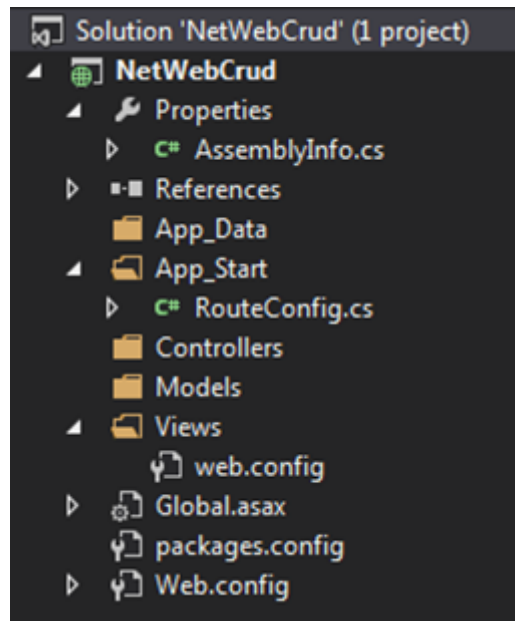


Figura 19: Estrutura de um projeto *ASP.NET MVC 5*

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 nossas páginas, e subpastas que contém páginas, devem ficar inseridas na pasta Views, todos os controllers devem ficar na pasta Controllers. No *ASP.NET MVC* existe o conceito de Areas, onde o desenvolvedor pode organizar melhor seu código, mas esse conceito não será 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 *AssemblyInfo* 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 será usado o banco de dados MySQL, essa pasta será apagada.

A pasta App_Start existe para que o desenvolvedor possa armazenar arquivos de inicialização da aplicação. Um novo projeto *ASP.NET MVC 5* vem com a classe *RouteConfig* onde a tabela de rotas que mapeia URLs para controllers é configurada. Esse arquivo será explicado no próximo capítulo quando será abordada 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 *AreaRegistration*.

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 aos projetos no Visual Studio. Nesse projeto serão 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 vai adicionar 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.

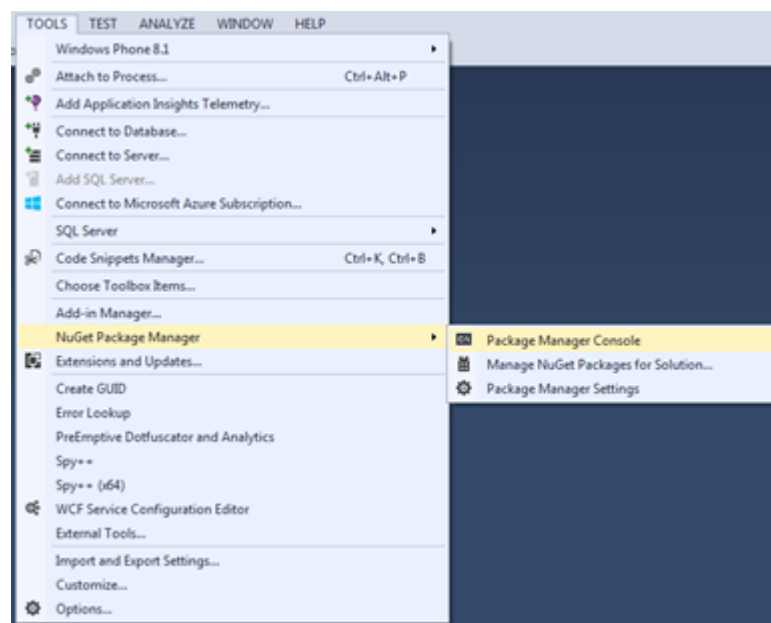


Figura 20: Acessando o console do NuGet

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.

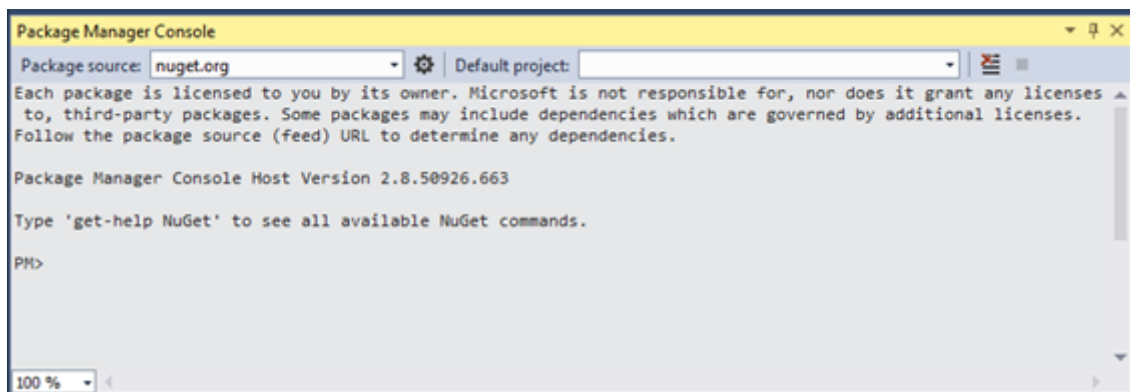


Figura 21: Console do NuGet

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 serão adicionado ao projeto. A figura 22 ilustra a instalação das bibliotecas.

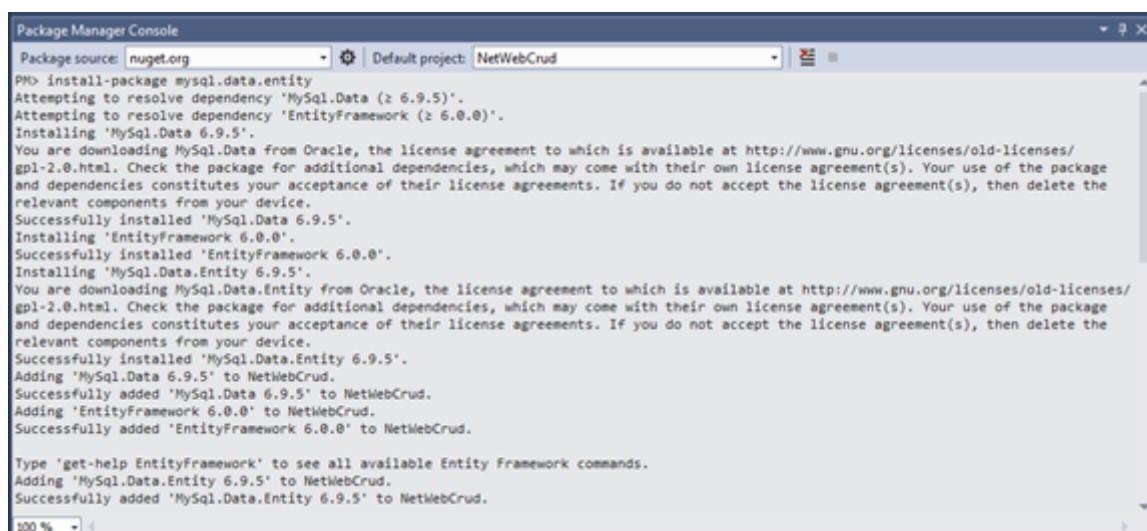


Figura 22: Console do NuGet

Ao terminar a instalação, os arquivos packages.config e Web.config serão 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 podemos observar o arquivo adicional citado. O conteúdo e funcionalidade desse arquivo será abordado no capítulo 5.

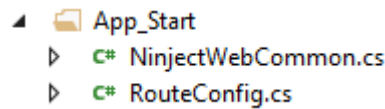


Figura 23: Arquivo adicional do Ninject

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 reflst: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>
  ...

```

Quadro 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 capítulo 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 capítulo será demonstrado como criar controllers e views em ambos os ambientes de desenvolvimento. Serão expostos exemplos de como criar controllers com ações que retornam páginas, objetos JSON e serão mostrados exemplos de ações que aceitam parâmetros. Serão 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 será criado um controller simples que irá retornar uma página estática para exibição no navegador.

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

Os exemplos desse capítulo, irão manipular uma classe chamada Person. Os qua-

dro 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;
    }
}
```

Quadro 7: Classe Person em Java

A anotação `@DateTimeFormat` especifica em qual formato 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; }
    }
}
```

Quadro 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";
    }
}
```

Quadro 9: PersonController no projeto Spring

A anotação `@RequestMapping` configura com qual o caminho, a partir da raiz da aplicação, as ações do controller serão acessadas. A mesma anotação em uma ação configura qual o caminho para acessá-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 específico da web. Então o caminho completo para o arquivo que será retornado é `<raiz do projeto>/WebContent/WEB-INF/person/add.jsp`.

Agora será 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.

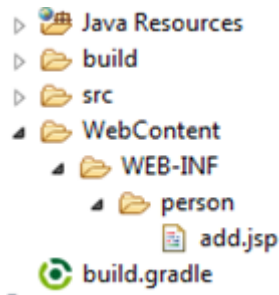


Figura 24: Primeira view adicionada ao projeto Java

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>
```

Quadro 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 do 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 selecionando as opções “Add” e “Controller”. A figura 25 ilustra a adição do controller.

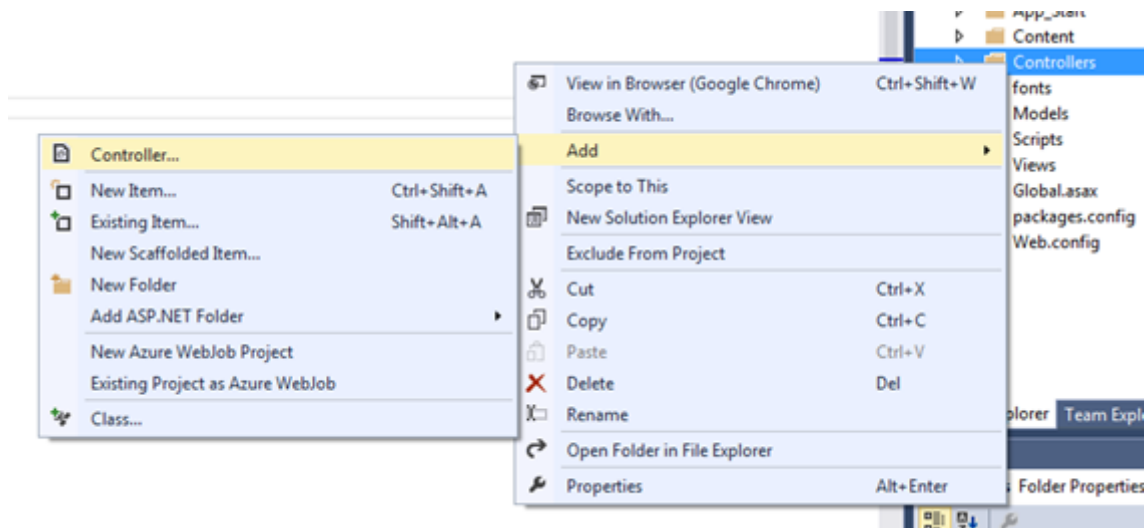


Figura 25: Adicionando um controller no Visual Studio

Irão aparecer 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 será adicionada uma ação chamada Add que irá retornar uma página. O código do controller deverá 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();
        }
    }
}
```

Quadro 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
                }
            );
        }
    }
}
```

Quadro 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">. Logo 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 intuito de apontar 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.

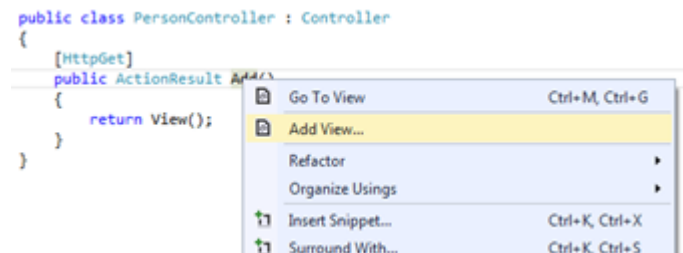


Figura 26: Adicionando uma View no ASP.NET MVC

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. Será 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 serão 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>

```

Quadro 13: O arquivo Add.cshtml

Tanto no projeto Java/Spring como no projeto ASP.NET MVC, quando acessada pelo Google Chrome, a view irá aparecer como na figura 28.

Add Person

Name:

Birth:

Figura 28: View adicionada ao projeto ASP.NET MVC

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 no controller que respondam ao mesmo endereço da página, mas usem método POST. Essas ações também deverão 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 serão feitas ações que receberão um objeto do tipo Person e o adicionarão 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";
}

```

Quadro 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.

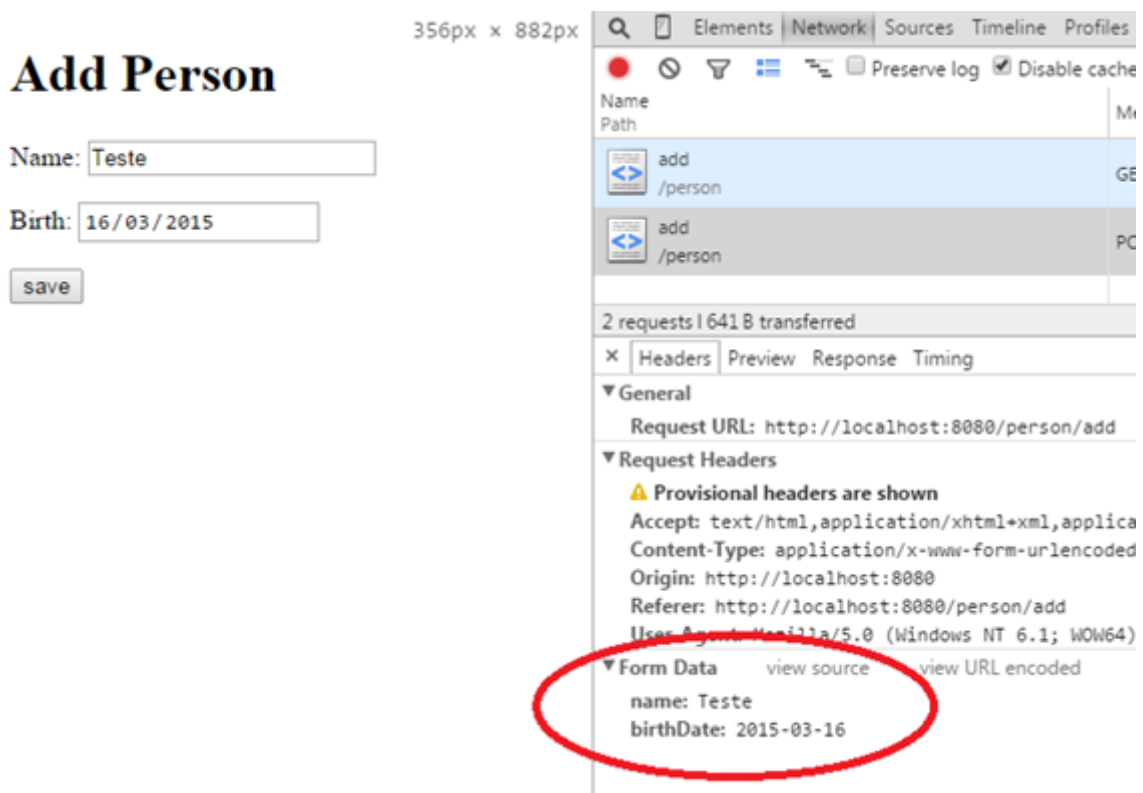


Figura 29: Enviando dados para o servidor

No quadro 7, o atributo `birthDate` da classe `Person` foi decorado com a anotação `@DateTimeFormat` para que aceitasse 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 outro padrão, o servidor enviaria uma resposta 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 trabalhar com os dados.

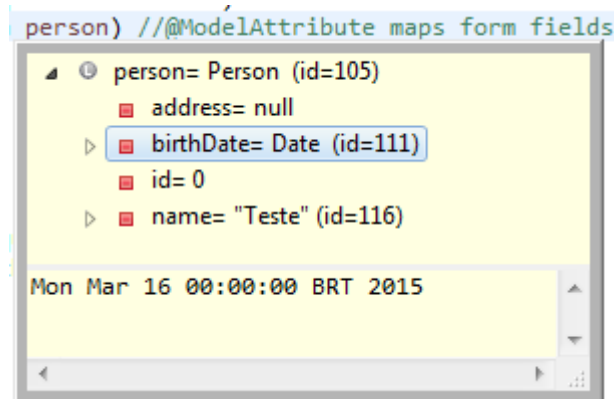


Figura 30: Objeto `person` com parâmetros corretamente populados no projeto Java

O retorno do método `addPost` redireciona a requisição para outra ação que mostrará uma página com uma tabela com todas pessoas adicionadas à lista. As implementações dessa ação e página serão demonstradas mais adiante.

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");
}

```

Quadro 15: PersonController do projeto ASP.NET com nova ação

O ASP.NET MVC não guarda o estado de objetos entre requisições, então a lista deverá ser guardada como uma variável da aplicação. Armazenada dessa forma, o estado da lista se preservará e ficará 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 responde 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 vai fazer 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.

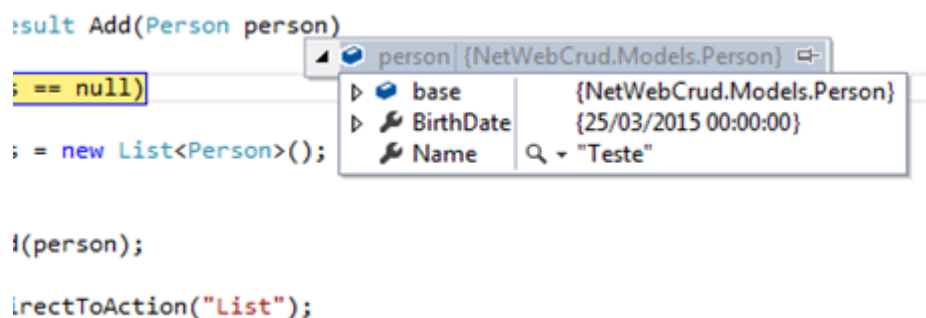


Figura 31: Objeto person no ASP.NET MVC corretamente populado

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 será demonstrado como retornar dados no formato JSON. Como exemplo será 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 serializar 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;
}
```

Quadro 16: Ação no Spring MVC que retorna um objeto JSON

Acessando a ação `getOne` pelo navegador, é obtida a resposta exposta na figura 32.

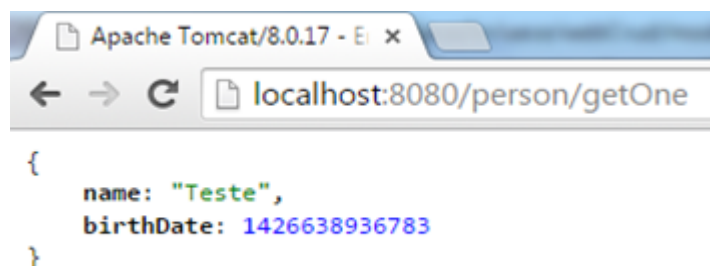


Figura 32: Resultado JSON no projeto Java

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

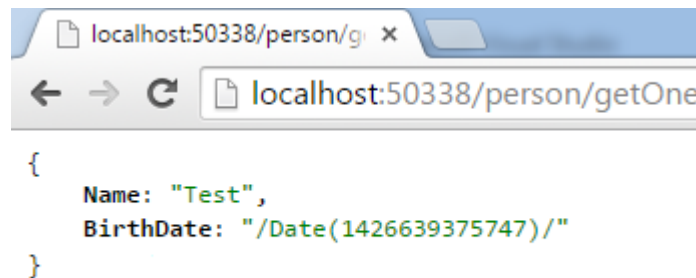
O ASP.NET MVC pode trabalhar com objetos no formato JSON sem precisar de uma biblioteca de terceiros. O exemplo do quadro 17 mostra uma ação que retorna um objeto JSON.

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

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

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

O método estático `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`.



```
{
  Name: "Test",
  BirthDate: "/Date(1426639375747)/"
}
```

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 será demonstrado como fazer uma página que exibe todos os itens das listas de objetos `Person` em ambos os projetos. Serão feitas ações que enviam as

listas para as views e irão montar o código HTML usando as view engines padrão de cada tecnologia, 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;
}
```

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

O tipo de retorno do método listPage é 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 referências 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>

```

Quadro 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 referências para passar a lista de objetos `Person` para a página. O atributo `var` configura um nome de referência para o objeto que será 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.

Person List

[Add Person](#)

Name	Birth Date
Teste	2015-03-20
Teste 2	2015-03-21
Teste 3	2015-03-22

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);
}
```

Quadro 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>

```

Quadro 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` pode fazer 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.

Person List

[Add Person](#)

Name	Birth Date
Teste	20/03/2015 00:00:00
Teste 2	21/03/2015 00:00:00
Teste 3	22/03/2015 00:00:00

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 específico para controllers objetivando uma melhor organização de código.

Ao enviar dados para o servidor, a única diferença significativa é o modo de enviar dados. 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 objeto 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`. Uma vantagem do ASP.NET MVC em relação ao Java Enterprise Edition em se tratando de views é o suporte nativo à `Layouts` (também conhecidas como `master pages`). Caso o desenvolvedor Java queira 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 às views, o desenvolvedor pode escolher simplesmente utilizar outras view engines.

No próximo capítulo serão abordados a criação de classes de negócio/serviços e injeção de dependências.

5 Classes de serviços e injeção de dependências

Uma aplicação 3-tier típica se divide em 3 camadas, 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 servem 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 será 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

Será criada como exemplo uma classe de serviço em cada projeto que conterá um método com uma regra de negócio simples. As listas de objetos Person, até agora pertencente aos controllers, serão movidas para essas classes como uma forma de representar a camada de persistência. Em ambos os projetos, o serviço irá implementar uma interface. Diferente do que é sugerido para controllers, que um controller não precisa 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 serão 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();
}
```

Quadro 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;
    }
}
```

Quadro 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 serão criadas dentro da pasta Models/Services. Para o ASP.NET MVC os modelos de uma aplicação 3-tier 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();
    }
}
```

Quadro 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;
    }
}
}

```

Quadro 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 serão injetadas nos controllers criados no capítulo 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();
    }
}

```

Quadro 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();
    }
}

```

Quadro 27: Exemplo de classes fracamente acopladas

No exemplo acima, a classe Foo agora recebe em seu construtor um objeto 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 serão os pacotes onde o Spring Framework irá procurar por classes para serem instanciadas e injetadas.

Com o objetivo de 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 irá procurar e instanciar 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 ?? 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";
    }
}

```

Quadro 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 irá procurar 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 será 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 a documentação do framework.

5.2.2 .NET

O ASP.NET MVC 5 não possui injeção de dependências automática nativa, então será usado a biblioteca Ninject para essa funcionalidade. Diferente do Spring, o Ninject não procura por classes de modo automático para injetar dependências, a classe que o desenvolvedor deseja injetar deverá ser configurada 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)
{
}
}

```

Quadro 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 ??.

```

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

```

Quadro 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 será 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 ??.

```
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());
        }
    }
}
```

Quadro 31: PersonController do projeto ASP.NET MVC usando PersonService