VRaptor 3 – Livro de receitas

http://vraptor.caelum.com.br

Automatically generated by Caelum Objects Tubaina
12 de novembro de 2009

Índice

1 Colocando objetos na sessão 1

2 ComponentFactory como seletor de implementações 2

3 Desabilitar exception do Page Result 3

4 Gerando aplicação com VRaptor3 usando Maven 4

5 Usando tiles com vraptor3 8

6 Interceptando recursos anotados 11

7 Poupando recursos - LAZY Dependency Injection 15

8 Evitando que o browser faça cache das páginas 19

9 Rodando o VRaptor3 no JBoss 21

10 Usando o Header Referer para fazer redirecionamentos 22

Version: 11.8.12

Colocando objetos na sessão

Se você precisa colocar algum objeto na Sessão (e está usando o Spring como DI provider) você pode criar um componente Session scoped:

```
@Component

@SessionScoped
public class UserInfo {

    private User user;

    // getter e setter
}
```

e se você quiser usar ou colocar o usuário na sessão, de dentro de algum Controller, por exemplo, você pode receber essa classe que você acabou de criar no construtor:

```
@Resource

public class LoginController {

    private UserInfo info;

    public LoginController(UserInfo info) {
        this.info = info;
    }

    //...
    public void login(User user) {
        //valida o usuario
        this.info.setUser(user);
    }
}
```

E para acessar esse usuário a partir da view, existe um atributo no HttpSession com o nome “userInfo”, assim você pode usar numa jsp:

```
${userInfo.user}
```

ComponentFactory como seletor de implementações

O uso típico para as ComponentFactories é para quando a dependência que você quer usar não faz parte do VRaptor nem da sua aplicação, como é o caso da Session do Hibernate.

Mas você pode usar as ComponentFactories para disponibilizar implementações de interfaces de acordo com alguma condição, por exemplo alguma configuração adicional.

Vamos supor que você quer fazer um enviador de Email, mas só quer usar o enviador de verdade quando o sistema estiver em produção, em desenvolvimento você quer um enviador falso:

```
public interface EnviadorDeEmail {

    void enviaEmail(Email email);
}

public class EnviadorDeEmailPadrao implements EnviadorDeEmail {
    //envia o email de verdade
}

public class EnviadorDeEmailFalso implements EnviadorDeEmail {
    //não faz nada, ou apenas loga o email
}
```

Sem anotar nenhuma dessas classes com @Component, você pode criar um ComponentFactory de EnviadorDeEmail, e anotá-lo com @Component:

```
@Component

@ApplicationScoped //ou @RequestScoped se fizer mais sentido
public class EnviadorDeEmailFactory implements ComponentFactory<EnviadorDeEmail> {

    private EnviadorDeEmail enviador;
    public EnviadorDeEmail(ServletContext context) {
        if ("producao".equals(context.getInitParameter("tipo.de.ambiente"))) {
            enviador = new EnviadorDeEmailPadrao();
        } else {
            enviador = new EnviadorDeEmailFalso();
        }

    }

    public EnviadorDeEmail getInstance() {
        return this.enviador;
    }
}
```

Desabilitar exception do Page Result

Quando você usa redirecionamentos para página:

```
result.use(Results.page()).forward(url);
```

e tenta passar uma URL que é tratada por alguma lógica da sua aplicação, o VRaptor vai lançar uma exceção falando para você usar o `result.use(logic())` correspondente. Por exemplo:

```
public class TesteController {

    @Path("/teste")
    public void teste() {}

    public void redireciona() {
        result.use(page()).redirect("/teste");
        // vai lançar uma exceção, falando para você usar o código
        // result.use(logic()).redirectTo(TesteController.class).teste();
    }
}
```

Se você quiser desabilitar esta exceção, você pode criar a seguinte classe:

```
@Component

public class MyPageResult extends DefaultPageResult {
    // delega o construtor
    @Override
    protected void checkForLogic(String url, HttpMethod httpMethod) {
       //nada aqui, ou algum outro tipo de checkagem
    }
}
```

Gerando aplicação com VRaptor3 usando Maven

por Lucas H. G. Toniazzo no blog *http://www.lucas.hgt.nom.br/wordpress/?p=107*

Algum tempo sem atualizar as coisas por aqui, resolvi tirar um tempo para me atualizar no framework que acompanho já faz algum tempo, VRAPTOR.

Ultimamente tenho utilizado o Maven para controlar as dependências de Jar's das aplicações que trabalho, e tem me ajudado bastante. Então resolvi tirar um tempo para gerar um POM para o VRaptor, já que na estrutura original dele esse arquivo não o acompanha.

Utilizei o Eclipse com suporte ao Maven para gerar o projeto após ter gerado o POM do VRaptor.

Alguns passos para se ter sucesso na criação da estrutura:

1) Faça o download do Vraptor e coloque o seu jar numa pasta qualquer.

2) Instale o Maven ou utilize o recurso que está disponível em sua IDE.

3) Execute o comando abaixo:

```
mvn install:install-file -DgroupId=br.com.caelum -DartifactId=vraptor

    -Dpackaging=jar -Dversion=3.0.0-SNAPSHOT
    -Dfile=LOCAL_ONDE_ESTA\vraptor3-3.0.0-SNAPSHOT.jar -DgeneratePom=true
```

4) Edite o pom.xml do VRaptor que deve estar no repositório local (`$USER_HOME/.m2/Repository/br/com/caelum/vraptor`) com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd"
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.caelum</groupId>
    <artifactId>vraptor</artifactId>
    <version>3.0.0-SNAPSHOT</version>
    <description>POM was created from install:install-file</description>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring</artifactId>
            <version>2.5.5</version>
```

```
</dependency>
<dependency>
    <groupId>com.thoughtworks.paranamer</groupId>
    <artifactId>paranamer</artifactId>
    <version>1.3</version>
</dependency>
<dependency>
    <groupId>org.objenesis</groupId>
    <artifactId>objenesis</artifactId>
    <version>1.1</version>
</dependency>
<dependency>
    <groupId>com.google.code.google-collections</groupId>
    <artifactId>google-collect</artifactId>
    <version>snapshot-20080530</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.6.0</version>
</dependency>
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib-nodep</artifactId>
    <version>2.2</version>
</dependency>
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.1</version>
    <exclusions>
        <exclusion>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.8.0.GA</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.1.2</version>
</dependency>
```

```
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
        <dependency>
            <groupId>net.vidageek</groupId>
            <artifactId>mirror</artifactId>
            <version>1.5.1</version>
        </dependency>
        <dependency>
            <groupId>ognl</groupId>
            <artifactId>ognl</artifactId>
            <version>2.7.3</version>
            <exclusions>
                <exclusion>
                    <groupId>jboss</groupId>
                    <artifactId>javassist</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.5.6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.5.6</version>
        </dependency>
    </dependencies>
</project>
```

5) Crie um projeto Maven.

6) Edite o pom.xml do mesmo com o seguinte conteúdo:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>GERALMENTE_ESTRUTURA_DO_PACKAGE</groupId>
    <artifactId>ARTIFACT_ID_QUE_DESEJAR</artifactId>
    <packaging>war</packaging>
    <name>NOME_QUE_DESEJAR</name>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>br.com.caelum</groupId>
            <artifactId>vraptor</artifactId>
            <version>3.0.0-SNAPSHOT</version>
```

```
        </dependency>
    </dependencies>
</project>
```

7) Clique com botão direito, opção Maven, update dependecies

Pronto, libs adicionadas ao projeto, tudo organizado para iniciar as atividades.

Usando tiles com vraptor3

por Otávio Scherer Garcia

Caso você queira integrar Tiles 2 com VRaptor fazendo com que seja feito forward diretamente para o tiles e não para o JSP (padrão no VRaptor3), basta escrever uma classe que implemente PathResolver com sua convenção.

A primeira coisa a fazer é colocar o tiles para responder como servlet. Assim toda requisição vinda por `*.tiles` será redirecionada ao tiles, e não ao VRaptor. Lembre-se de colocar a declaração do tiles servlet antes do vraptor filter.

web.xml:

```
<!-- arquivo de definições do tiles -->
<context-param>
 <param-name>org.apache.tiles.impl.BasicTilesContainer.DEFINITIONS_CONFIG</param-name>
  <param-value>/WEB-INF/classes/tiles.xml</param-value>
</context-param>

<!-- servlet de inicialização do tiles -->
<servlet>
  <servlet-name>TilesServlet</servlet-name>
  <servlet-class>org.apache.tiles.web.startup.TilesServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>

<!-- servlet que responde as requisições do tiles -->
<servlet>
  <servlet-name>TilesDispatchServlet</servlet-name>
 <servlet-class>org.apache.tiles.web.util.TilesDispatchServlet</servlet-class>
</servlet>

<!-- o tiles responderá por toda requisição *.tiles -->
<servlet-mapping>
  <servlet-name>TilesDispatchServlet</servlet-name>
  <url-pattern>*.tiles</url-pattern>
</servlet-mapping>
```

No meu caso usei como padrão para o tiles a convenção /package.controller.metodo. Sendo assim criei o seguinte path resolver.

```
@Component

public class TilesPathResolver
  implements PathResolver {

  static final String VIEW_SUFIX = ".tiles";
```

```
  static final String CLASS_SUFIX = "Controller";

  @Override
  public String pathFor(ResourceMethod method) {
      final Class<?> clazz = method.getResource().getType();

      final StringBuilder s = new StringBuilder();
      s.append("/");
      // retorna apenas o nome do último pacote
      s.append(StringUtils.substringAfterLast(clazz.getPackage().getName(), "."));
      s.append(".");
      //remove o sufixo controller
      s.append(StringUtils.substringBefore(clazz.getSimpleName(), CLASS_SUFIX));
      s.append(".");
      s.append(method.getMethod().getName());
      s.append(VIEW_SUFIX);

   // definições do tile em minusculo, mas você pode alterar isso
      return s.toString().toLowerCase();
  }
}
```

Se você não quiser utilizar a classe `StringUtils` do projeto `commons-lang`, você pode alterar o metodo para:

```
final Class<?> clazz = method.getResource().getType();

String pkgname = clazz.getPackage().getName();

final StringBuilder s = new StringBuilder();
s.append("/");
// retorna apenas o nome do último pacote
s.append(pkgname.substring(pkgname.lastIndexOf(".") + 1));
s.append(".");
//remove o sufixo controller
s.append(clazz.getSimpleName().substring(0, clazz.getSimpleName().indexOf(CLASS_SUFIX)));
s.append(".");
s.append(method.getMethod().getName());
s.append(VIEW_SUFIX);

// definições do tile em minusculo, mas você pode alterar isso
return s.toString().toLowerCase();
```

Nesse caso se você tiver o controler `CustomerController` e chamar o método `list`, a view será redirecionada ao URI /admin.customer.list.tiles, que será executado pelo tiles servlet.

```
package xpto.admin;


@Resource
public class CustomerController {
  public List<Customer> list() {
      return myServiceClass.listAllCustomers();
      // será redirecionado para a definição admin.customer.list
  }
}


<definition name="admin.customer.list" extends="default">
```

```
  <put-attribute name="body" value="/WEB-INF/jspx/admin/customer.list.jspx" />
</definition>
```

Referências:

- **Apache Tiles**: http://tiles.apache.org/
- **VRaptor View e PathResolver**: http://vraptor.caelum.com.br/documentacao/view-e-ajax/

Interceptando recursos anotados

por Tomaz Lavieri, no blog *http://blog.tomazlavieri.com.br/2009/vraptor3-interceptando-recursos-anotados/*

Em primeira lugar, gostaria de tecer meus mais sinceros elogios a equipe VRaptor, a versão 3 esta muito boa, bem mais intuitiva e fácil de usar que a 2.6

Neste artigo vou mostar como interceptar um método de um Resource especifico, identificando-o a partir de uma anotação e executar ações antes do método executar, e após ele executar.

Vamos supor que nós temos o seguinte Resource para adicionar produtos no nosso sistema

```
@Resource

public class ProdutoController {
    private final DaoFactory factory;
    private final ProdutoDao dao;

    public ProdutoController(DaoFactory factory) {
        this.factory = factory;
        this.dao = factory.getProdutoDao();
    }

    public List<Produto> listar() {
        return dao.list();
    }

    public Produto atualizar(Produto produto) {
        try {
            factory.beginTransaction();
            produto = dao.update(produto);
            factory.commit();
            return produto;
        } catch (DaoException ex) {
            factory.rollback();
            throw ex;
        }
    }

    public Produto adicionar(Produto produto) {
        try {
            factory.beginTransaction();
            produto = dao.store(produto);
            factory.commit();
            return produto;
        } catch (DaoException ex) {
            factory.rollback();
            throw ex;
        }
    }
```

```
}
```

Agora nos queremos que um interceptador intercepte meu recurso, e execute a lógica dentro de um escopo transacional, como fazer isso? é só criar um interceptador assim.

```
import org.hibernate.Session;

import org.hibernate.Transaction;

import br.com.caelum.vraptor.Intercepts;
import br.com.caelum.vraptor.core.InterceptorStack;
import br.com.caelum.vraptor.interceptor.Interceptor;
import br.com.caelum.vraptor.resource.ResourceMethod;

@Intercepts
public class TransactionInterceptor implements Interceptor {
        private final Session session;
        public HibernateTransactionInterceptor(Session session) {
            this.session = session;
        }
        public void intercept(InterceptorStack stack, ResourceMethod method,
                              Object instance) {
            Transaction transaction = null;
            try {
                transaction = session.beginTransaction();
                stack.next(method, instance);
                transaction.commit();
            } finally {
                if (transaction.isActive()) {
                    transaction.rollback();
                }
            }
        }
        public boolean accepts(ResourceMethod method) {
            return true; //aceita todas as requisições
        }
}
```

Ok, o interceptador vai rodar e abrir transação antes e depois de executar a logica, e os métodos transacionais da minha lógica irão se reduzir a isto

```
public Produto atualizar(Produto produto) {

    return dao.update(produto);
}

public Produto adicionar(Produto produto) {
    return dao.store(produto);
}
```

Ok mas, neste caso temos o problema de que métodos que não exigem transação estão abrindo e fechando transação a cada requisição sem necessidade.

Como então selecionar apenas algumas lógicas para serem transacionais? podem criar uma anotação para isto, desta forma:

```
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Usado para garantir que um determinado recurso interceptado seja executada em um
 * escopo de transação.
 * @author Tomaz Lavieri
 * @since 1.0
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface Transactional {}
```

Agora precisamos marcar os pontos onde queremos que o escopo seja transacional com esta anotação.

```
 @Resource

public class ProdutoController {
    private final ProdutoDao dao;

    public ProdutoController(ProdutoDao dao) {
        this.dao = dao;
    }

    public List<Produto> listar() {
        return dao.list();
    }

    @Transactional
    public Produto atualizar(Produto produto) {
        return dao.update(produto);
    }

    @Transactional
    public Produto adicionar(Produto produto) {
        return dao.store(produto);
    }
}
```

Ok o código ficou bem mais enxuto, mas como interceptar apenas os métodos marcados com esta anotação?? para tal basta no nosso accepts do TransactionInterceptor verificarmos se a anotação esta presente no método, ou no proprio rescurso (quando marcado no recurso todos os métodos do recurso seriam transacionais).

A modificação do método ficaria assim

```
 public boolean accepts(ResourceMethod method) {

    return  method
                .getMethod() //metodo anotado
                .isAnnotationPresent(Transactional.class)
            || method
                .getResource() //ou recurso anotado
                .getType()
                .isAnnotationPresent(Transactional.class);
}
```

Pronto agora somente os métodos com a anotação @Transacional são executados em escopo de transação e economisamos linhas e linhas de códigos de `try{commit}catch{rollback throw ex}`

Poupando recursos - LAZY Dependency Injection

por Tomaz Lavieri no blog *http://blog.tomazlavieri.com.br/2009/vraptor-3-poupando-recursos-lazy-dependence-injection/*

Objetivo: Ao final deste artigo espera-se que você saiba como poupar recursos caros, trazendo eles de forma *LAZY*, ou como prefiro chamar *Just-in-Time* (no momento certo).

No VRaptor3 a injeção de dependência ficou bem mais fácil, os interceptadores que eram os responsáveis para injetar a dependência sumiram e agora fica tudo a cargo do container, que pode ser o Spring ou o Pico.

A facilidade na injeção de dependencia tem um custo, como não é mais controlado pelo programador que cria o interceptor sempre que declaramos uma dependência no construtor de um `@Component`, `@Resource` ou `@Intercepts` ele é injetado no início, logo na construção, porém às vezes o fluxo de um requisição faz com que não usemos algumas destas injeções de dependência, disperdiçando recursos valiosos.

Por exemplo, vamos supor o seguinte `@Resource` abaixo, que cadastra produtos

```
import java.util.List;

import org.hibernate.Session;
import br.com.caelum.vraptor.Result;
import br.com.caelum.vraptor.view.Results;

@Resource
public class ProdutoController {
    /**
     * O recurso que queremos poupar.
     */
    private final Session session;
    private final Result result;

    public ProdutoController(final Session session, final Result result) {
        this.session = session;
        this.result = result;
    }

    /**
     * apenas renderiza o formulário
     */
    public void form() {}

    public List<Produto> listar() {
        return session.createCriteria(Produto.class).list();
    }

    public Produto adiciona(Produto produto) {
        session.persist(produto);
```

```
        result.use(Results.logic()).redirectTo(getClass()).listar();
        return produto;
    }
}
```

Sempre que alguem faz uma requisição a qualquer lógica dentro do recurso `ProdutoController` uma `Session` é aberta, porem note que abrir o formulário para adicionar produtos não requer sessão com o banco, ele apenas renderiza uma página, cada vez que o formulário de produtos é aberto um importante e caro recurso do sistema esta sendo requerido, e de forma totalmente ociosa.

Nota do editor

É criada no máximo uma Session por requisição. A mesma coisa para qualquer componente Request scoped.

Como agir neste caso? Isolar o formulário poderia resolver este problema mais recairia em outro, da mantenabilidade.

O ideal é que este recurso só fosse realmente injetado no tempo certo (*Just in Time*) como seria possível fazer isso? A solução é usar proxies dinâmicos, enviando uma `Session` que só realmente abrirá a conexão com o banco quando um de seus métodos for invocado.

```
import java.lang.reflect.Method;

import javax.annotation.PreDestroy;
import org.hibernate.classic.Session;
import org.hibernate.SessionFactory;
import net.vidageek.mirror.dsl.Mirror;
import br.com.caelum.vraptor.ioc.ApplicationScoped;
import br.com.caelum.vraptor.ioc.Component;
import br.com.caelum.vraptor.ioc.ComponentFactory;
import br.com.caelum.vraptor.ioc.RequestScoped;
import br.com.caelum.vraptor.proxy.MethodInvocation;
import br.com.caelum.vraptor.proxy.Proxifier;
import br.com.caelum.vraptor.proxy.SuperMethod;

/**
 * <b>JIT (Just-in-Time) {@link Session} Creator</b> fábrica para o
 * componente {@link Session} gerado de forma LAZY ou JIT(Just-in-Time)
 * a partir de uma {@link SessionFactory}, que normalmente se encontra
 * em um ecopo de aplicativo @{@link ApplicationScoped}.
 *
 * @author Tomaz Lavieri
 * @since 1.0
 */
@Component
@RequestScoped
public class JITSessionCreator implements ComponentFactory<Session> {

    private static final Method CLOSE =
            new Mirror().on(Session.class).reflect().method("close").withoutArgs();
    private static final Method FINALIZE =
            new Mirror().on(Object.class).reflect().method("finalize").withoutArgs();

    private final SessionFactory factory;
    /** Guarda a Proxy Session */
```

```
    private final Session proxy;
    /** Guarada a Session real. */
    private Session session;

    public JITSessionCreator(SessionFactory factory, Proxifier proxifier) {
        this.factory = factory;
        this.proxy = proxify(Session.class, proxifier); // *1*
    }

    /**
     * Cria o JIT Session, que repassa a invocação de qualquer método, exceto
     * {@link Object#finalize()} e {@link Session#close()}, para uma session real,
     * criando uma se necessário.
     */
    private Session proxify(Class<? extends Session> target, Proxifier proxifier) {
        return proxifier.proxify(target, new MethodInvocation<Session>() {
            @Override // *2*
            public Object intercept(Session proxy, Method method, Object[] args,
                                                                SuperMethod superMethod) {
                if (method.equals(CLOSE)
                        || (method.equals(FINALIZE) && session == null)) {
                    return null; //skip
                }
                return new Mirror().on(getSession()).invoke().method(method)
                                   .withArgs(args);
            }
        });
    }

    public Session getSession() {
        if (session == null) // *3*
                session = factory.openSession();
        return session;
    }

    @Override
    public Session getInstance() {
        return proxy; // *4*
    }

    @PreDestroy
    public void destroy() { // *5*
        if (session != null && session.isOpen()) {
            session.close();
        }
    }
}
```

Explicando alguns pontos chaves, comentados com `// *N*`

- O `Proxfier` é um objeto das libs do VRaptor que auxilia na criação de objetos proxys ele é responsável por escolher a biblioteca que implementa o proxy dinâmico, e então invocar via callback um método interceptor, como falamos abaixo.
- Neste ponto temos a implementação do nosso interceptor, sempre que um método for invocado em nosso proxy, esse intereptor é invocado primeiro, ele filtra as chamadas ao método `finalize` caso a `Session` real ainda não tenha sido criada, isso evita criar a `Session` apenas para finaliza-la. O método `close` também

é filtrado, isso é feito para evitar criar uma session apenas para fechá-la, e também por que o nosso `SessionCreator` é que é o responsavel por fechar a session ao final do scopo, quando a request acabar. Todos os outros métodos são repassados para uma session através do método `getSession()` onde é realmente que acontece o `LAZY` ou `JIT`.

- Aqui é onde acontece a mágica, da primeira vez que `getSession()` é invocado a sessão é criada, e então repassada, todas as outras chamadas a `getSession()` repassam a sessão anteriormente criada, assim, se `getSession()` nunca for invocado, ou seja, se nenhum método for invocado no proxy, `getSession()` nunca será invocado, e a sessão real não será criada.
- O retorno desse `ComponentFactory` é a `Session` proxy, que só criará a session real se um de seus métodos for invocado.
- Ao final do escopo o `destroy` é invocado, ele verifica se a session real existe, existindo verifica se esta ainda esta aberta, e estando ele fecha, desta forma é possivel garantir que o recurso será sempre liberado.

Assim podemos agora pedir uma Session sempre que acharmos que vamos precisar de uma, sabendo que o recurso só será realmente solicitado quando formos usar um de seus métodos, salvando assim o recurso.

Esta mesma abordagem pode ser usada para outros recursos caros do sistema.

Os códigos fonte para os `ComponentFactory` de `EntityManager` e `Session` que utilizo podem ser encontrados neste link: http://guj.com.br/posts/list/141500.java

Evitando que o browser faça cache das páginas

por Otávio Scherer Garcia

Este interceptor indica ao browser para não efetuar cache das páginas.

```
/**

 * Intercepts all requests and set a no-cache information.
 *
 * @author Otávio Scherer Garcia
 * @version $Revision$
 */
@Intercepts
@RequestScoped
public class NoCacheInterceptor
    implements Interceptor {

    private final HttpServletResponse response;

    public NoCacheInterceptor(HttpServletResponse response) {
        this.response = response;
    }

    @Override
    public boolean accepts(ResourceMethod method) {
        return true; // allow all requests
    }

    @Override
    public void intercept(InterceptorStack stack, ResourceMethod method,
                Object resourceInstance)
        throws InterceptionException {
        // set the expires to past
        response.setHeader("Expires", "Wed, 31 Dec 1969 21:00:00 GMT");

        // no-cache headers for HTTP/1.1
        response.setHeader("Cache-Control", "no-store, no-cache, must-revalidate");

        // no-cache headers for HTTP/1.1 (IE)
        response.addHeader("Cache-Control", "post-check=0, pre-check=0");

        // no-cache headers for HTTP/1.0
        response.setHeader("Pragma", "no-cache");

        stack.next(method, resourceInstance);
    }
}
```

Rodando o VRaptor3 no JBoss

Existe um bug no Spring 2.5.x https://jira.jboss.org/jira/browse/SNOWDROP-4, usado por padrão no VRaptor3, que impede o escaneamento de classpath dentro do JBoss. Por isso, o VRaptor não consegue identificar as classes da sua aplicação.

Para corrigir isso, baixe a versão mais nova do Spring em http://www.springsource.com/download, que não contém esse bug.

Com a versão mais nova do spring em mãos, remova o spring-2.5.x.jar do seu WEB-INF/lib, e adicione os jars da nova versão do Spring.

Usando o Header Referer para fazer redirecionamentos

Geralmente quando você clica em um link, ou submete um formulário, o browser envia uma requisição para o servidor da sua aplicação, colocando um Header chamado Referer, que contém qual é a página atual, que originou a requisição.

Você pode usar esse Header com o VRaptor, para fazer os redirecionamentos:

```
import static br.com.caelum.vraptor.view.Results.referer;

@Resource
public class ShoppingController {
    //...
    public void adicionaItem(Item item) {
        validator.checking(...);
        validator.onErrorUse(referer()).forward();

        dao.adiciona(item);

        result.use(referer()).redirect();
    }
}
```

O problema em usar o Referer é que ele não é obrigatório. Então quando o Referer não vem na requisição, o VRaptor vai lançar uma IllegalStateException, e assim você pode especificar uma outra lógica para ir caso o Referer não seja especificado:

```
try {

    result.use(referer()).redirect();
} catch (IllegalStateException e) {
    result.use(logic()).redirectTo(HomeController.class).index();
}
```