VRaptor

http://vraptor.caelum.com.br

Automatically generated by Caelum Objects Tubaina
24 de fevereiro de 2012

Índice

Version: 14.5.24

Aceitando URLs com ou sem barra no final

por Tomaz Lavieri

Para quem teve dificuldade como eu em conseguir determinar urls como

```
@Path("produto/{produto.id}")
```

quando digitava a URI "/produto/1/" e o link não funcionava, segue abaixo uma modificação que corrige o problema.

> **Nota do editor**
>
> Isso não é necessariamente um problema... a URL /abc é diferente da /abc/ portanto o comportamento de dar 404 é o esperado. Mas se você quiser que sejam urls equivalentes você pode escrever a classe abaixo.

```
import br.com.caelum.vraptor.Result;
import br.com.caelum.vraptor.core.RequestInfo;
import br.com.caelum.vraptor.http.route.ResourceNotFoundException;
import br.com.caelum.vraptor.http.route.Router;
import br.com.caelum.vraptor.ioc.Component;
import br.com.caelum.vraptor.ioc.RequestScoped;
import br.com.caelum.vraptor.resource.DefaultResourceNotFoundHandler;
import br.com.caelum.vraptor.resource.HttpMethod;
import br.com.caelum.vraptor.view.Results;

@Component
@RequestScoped
public class Error404 extends DefaultResourceNotFoundHandler  {
    private final Router router;
    private final Result result;
    public Error404(Router router, Result result) {
        this.router = router;
        this.result = result;
    }

    @Override
    public void couldntFind(RequestInfo requestInfo) {
        try {
            String uri = requestInfo.getRequestedUri();
            if (uri.endsWith("/")) {
```

```
                tryMovePermanentlyTo(requestInfo, uri.substring(0, uri.length()-1));
            } else {
                tryMovePermanentlyTo(requestInfo, uri + "/");
            }
        } catch (ResourceNotFoundException ex) {
            super.couldntFind(requestInfo);
        }
    }

    private void tryMovePermanentlyTo(RequestInfo requestInfo, String newUri) {
        router.parse(newUri, HttpMethod.of(requestInfo.getRequest()),
            requestInfo.getRequest());
        result.use(Results.http()).movedPermanentlyTo(newUri);
    }
}
```

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}`

Job Scheduling com vRaptor3 e Spring

Por Mário Peixoto no blog *http://mariopeixoto.wordpress.com/2009/11/17/job-scheduling-com-vraptor3-e-spring3/*

Este artigo tem como objetivo mostrar um caminho simples para fazer job scheduling no vRaptor3 quando o Spring for o seu container IoC sem que seja necessário o uso de XML para configuração.

Versões utilizadas:

- vRaptor 3.0.2
- Spring 3.0.0.RC2

8.1- O Task Scheduler

O Spring já vem com suporte a task scheduling/executing, ele precisa apenas que uma implementação de `org.springframework.scheduling.TaskScheduler` esteja disponível no escopo de aplicação (*singleton* no Spring). Para isso precisamos implementar um `ComponentFactory`:

```
@Component
@ApplicationScoped
public class TaskSchedulerFactory implements ComponentFactory<TaskScheduler> {

    private ThreadPoolTaskScheduler scheduler;

    @PostConstruct
    public void create() {
        scheduler = new ThreadPoolTaskScheduler();
        scheduler.initialize();
    }

    public TaskScheduler getInstance() {
        return scheduler;
    }

    @PreDestroy
    public void destroy() {
        scheduler.destroy();
    }

}
```

Neste exemplo utilizaremos o `ThreadPoolTaskScheduler` como implementação para o `TaskScheduler`, porém várias outras classes vem com o Spring e podem ser usadas da mesma maneira (documentação).

8.2- As Tasks/Jobs

Vamos criar interfaces para nossas tasks.

```
public interface Task extends Runnable {
    void schedule(TaskScheduler scheduler);
}

public interface ApplicationTask extends Task {
}

public interface RequestTask extends Task {
}
```

Registrando tasks em escopo de application

Para registrar esse tipo de task, precisaremos de um bean em escopo de application:

```
@Component
@ApplicationScoped
public class MyFirstApplicationTask implements ApplicationTask {
    public MyFirstApplicationTask(TaskScheduler scheduler) {
        //Aqui você poderá receber componentes que não estejam em
        //escopo de request ou session
        ...
        this.schedule(scheduler);
    }
    public void schedule(TaskScheduler scheduler) {
        scheduler.schedule(this, new CronTrigger("0 0 23 * * *"));
        //Neste caso, a task rodará sempre às 23h0min0s
    }
}
```

Registrando tasks em escopo de request

Para registrar esse tipo de task, basta receber o scheduler no construtor do seu `Resource` e registrar a task no método adequado.

```
@Component
@RequestScoped
public class MyFirstRequestTask implements RequestTask {
    public MyFirstApplicationTask() {
        //Aqui você poderá receber qualquer componente que precisar
        ...
    }
    public void schedule(TaskScheduler schedule) {
        Calendar now = GregorianCalendar.getInstance();
        now.add(Calendar.MINUTE,5);
        scheduler.schedule(this, now.getTime());
        //Neste caso, a task rodará apenas uma vez, 5 min depois da
        //execução deste método
    }
}
@Resource
public class MyResource {
    private MyFirstRequestTask task;
```

```
    private TaskScheduler scheduler;
    public MyResource(MyFirstRequestTask task, TaskScheduler scheduler) {
        this.task = task;
        this.scheduler = scheduler;
    }
    ...
    public void taskInit() {
        task.schedule(scheduler);
    }
    ...
}
```

8.3- Observações

Criamos 3 interfaces, pois deste modo ficará mais fácil injetar diferentes tipos de tasks no construtor, por exemplo podemos receber todas as tasks em escopo de request dentro de um resource assim:

```
@Resource
public class MyResource {
    private List<RequestTask> tasks;
    private TaskScheduler scheduler;
    public MyResource(List<RequestTask> tasks, TaskScheduler scheduler) {
        this.tasks = tasks;
        this.scheduler = scheduler;
    }
    ...
    public void tasksInit() {
        for (RequestTask task : tasks) {
            task.schedule(scheduler);
        }
    }
    ...
}
```

8.4- O escopo Prototype

O Spring possui um escopo chamado prototype que define que o componente terá uma nova instância sempre que for requisitada, este escopo ainda não existe no vRaptor3 mas será implementado em breve. Este escopo servirá para definir compenentes utilizados por tasks em escopo de application que naturalmente não são singletons, como DAO's.

Você não poderá receber a Session no construtor do seu DAO prototyped pois geralmente a SessionFactory está em escopo de request, e o vRaptor ainda não possui suporte às scoped-proxies do Spring. Então precisará fazer assim:

```
@Component
@PrototypeScoped
public class MyPrototypedDAO extends ... implements ... {

    private Session session;

    public MyPrototypedDAO(SessionFactory sessionFactory) {
        this.session = sessionFactory.openSession();
    }
```

```
    ...
}
```

Deste modo, será criado uma instância do seu DAO para cada requisição de instância (lê-se: sempre que você receber uma instância pelo construtor, será uma nova), assim como uma Session separada.

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 e Lucas Cavalcanti

Este interceptor indica ao browser para não efetuar cache das páginas, adicionando headers indicando que a página atual está expirada.

```
@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);
    }
}
```

Você pode também criar uma anotação para que apenas as classes anotadas evitem o cache.

```
@Target({ElementType.METHOD, ElementType.TYPE})
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface NoCache {

}
```

E alterar o método accepts do interceptor para o código abaixo:

```
 @Override

public boolean accepts(ResourceMethod method) {
    return method.containsAnnotation(NoCache.class);
}
```

Usando Objectify com VRaptor

por Otávio Scherer Garcia

Para trabalhar de forma simples no VRaptor com a Objectify é necessário criar uma classe que inicie a instância da ObjectifyFactory. Como precisamos apenas de uma única instância esta classe pode possuir escopo de aplicação:

```
@Component

@ApplicationScoped
public class ObjectifyFactoryCreator
    implements ComponentFactory<ObjectifyFactory> {

    private final ObjectifyFactory instance = new ObjectifyFactory();

    @PostConstruct
    public void create() {
        instance.register(MyFirstEntity.class);
        instance.register(MySecondEntity.class);
    }

    public ObjectifyFactory getInstance() {
        return instance;
    }
}
```

Após criamos a classe que vai ser responsável por criar as intâncias de Objectify. Como precisamos de uma única instância para cada requisição, o escopo deste componente será request.

```
@Component

@RequestScoped
public class ObjectifyCreator
    implements ComponentFactory<Objectify> {

    private final ObjectifyFactory factory;
    private Objectify ofy;

    public ObjectifyCreator(ObjectifyFactory factory) {
        this.factory = factory;
    }

    @PostConstruct
    public void create() {
        ofy = factory.begin();
    }

    public Objectify getInstance() {
        return ofy;
```

```
    }
}
```

Para utilizar a Objectify basta injetá-la no construtor conforme o exemplo abaixo:

```
public class MyController {

    private final Objectify ofy;

    public MyController(Objectify ofy) {
        this.ofy = ofy;
    }

    public void anything() {
        ofy.get(new Key<MyFirstEntity>(MyFirstEntity.class, 1L));
    }
}
```

Usando o Static Scanning no GAE/J

por Otávio Scherer Garcia

Uma das formas para diminuir o tempo de inicialização do VRaptor no GAE/J é utilizar o Static Scanning. Com esta funcionalidade o VRaptor busca pelos componentes no tempo de build. Desta forma o Scanning Dinâmico não é necessário.

Basta adicionar as seguintes linhas no seu `build.xml` e logo após incluir esta task como dependência da task `update`.

```
<target name="vraptor-scanning" depends="compile">
    <path id="build.classpath">
        <fileset dir="war/WEB-INF/lib" includes="*.jar" />
    </path>

    <java classpathref="build.classpath"
            classname="br.com.caelum.vraptor.scan.VRaptorStaticScanning" fork="true">
        <arg value="war/WEB-INF/web.xml" />
        <classpath refid="build.classpath" />
        <classpath path="war/WEB-INF/classes" />
    </java>
</target>
```

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();
}
```

Configurando o VRaptor na Locaweb

por Flavio Gabriel Duarte

Este tutorial tem como objetivo orientar o deploy de sistemas usando o framework VRaptor na Locaweb com JVM decidado em ambiente Linux compartilhado.

Preparando o ambiente:

Toda a requisição que chega até a Locaweb vai primeiramente para o Apache e se necessário no mod_jk (http://en.wikipedia.org/wiki/Mod_jk) passará enviará a requisição para o Tomcat, como explicado nessa página http://wiki.locaweb.com.br/pt-br/Tomcat_integrado_com_o_Apache

A configuração sugerida pela Locaweb leva em conta que a sua aplicação não usa nenhum framework, apenas arquivos jsps e servlets configurado pelo arquivo web.xml.

Como o VRaptor não usa o web.xml para definir as urls a configuração sugerida pela Locaweb não funcionará. Para funcionar você precisa abrir um chamado na Locaweb informando que gostaria de usar o framework e escrever a seguinte regra: JkMount /* [login] >> TOMCAT e a pasta do servidor que deseja aplicar a regra caso não seja a raiz.

Aguarde a resposta do chamado para saber quando estará funcionando. O procedimento leva +- 24h para eles realizarem, então peça com antecedência.

Após isso basta enviar a sua aplicação para a Locaweb e reinicar o ambiente que ela estará funcionando. A Locaweb não aceita deploy com arquivos WAR, para enviar os arquivos pelo WAR você precisaria enviá-lo pelo FTP e depois descompactar pelo SSH. Se você está usando o eclipse para desenvolver a sua aplicação e o recurso “servers” do próprio eclipse, a sua aplicação fica dentro de workspace/.metadata/.plugins/org.eclipse.wst.server.core. Dentro deve ter algumas pasta “tmp” isso vai de acordo com a quantidade de servidores que você tenha, alguma delas deve ser a sua aplicação. Como a aplicação está disposta dentro desta pasta, ela poderia ir para a locaweb e ser executada. Essa não é a melhor de fazer o deploy, mas é uma das aceitas neste tipo de ambiente na Locaweb.

A Locaweb define um timeout de 15 segundos em suas conexões MySQL, se você tiver usando este banco e não atentar para isso terá problemas, para contornar essa problema você pode seguir a sugestão da Locaweb http://wiki.locaweb.com.br/pt-br/Resolvendo_Problemas_Conex%C3%A3o_JAVA_com_MYSQL ou configurar o c3p0 como descrito nesse post http://blog.caelum.com.br/2009/10/19/a-java-net-socketexception-broken-pipe/ respeitando o tempo imposto pela Locaweb.

VRaptor e Tiles 2.2

por Rogerio Alcantara em *http://www.guj.com.br/posts/list/215206.java#1098196*

baixar o tiles-jsp.jar. Como estou utilizando maven2:

```
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-jsp</artifactId>
    <version>2.2.2</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

No web.xml:

```
<!-- tiles configuration -->
<servlet>
    <servlet-name>TilesServlet</servlet-name>
    <servlet-class>org.apache.tiles.web.startup.TilesServlet</servlet-class>
    <init-param>
        <param-name>org.apache.tiles.factory.TilesContainerFactory.MUTABLE</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<!-- /tiles configuration -->

<!-- vraptor configuration -->
<filter>
    <filter-name>vraptor</filter-name>
    <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
</filter>

<filter-mapping>
    <filter-name>vraptor</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
<!-- vraptor configuration -->
```

Repare que não é declarado o DEFINITIONS_CONFIG, nem registrado o TilesDispatchServlet. Pois não utilizaremos arquivo para guardar as definitions, e quem cuidara dos redirecionamentos continuará sendo o VRaptor, que já faz isso muito bem obrigado! ;D Outro detalhe importante, é deixar o TilesContainerFactory como MUTABLE. ;)

TilesPathResolver?

Nessa abordagem, não é necessário criar implementar esse PathResolver, já que essa responsabilidade continuará sendo do VRaptor. ;D

/WEB-INF/jsp/template.jsp - servirá de base para todas as páginas

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>site</title>
    </head>
    <body>
        <div id="divHeader">
            <tiles:insertTemplate template="/WEB-INF/jsp/header.jsp"/>
        </div>
        <div id="divContent">
            <tiles:insertAttribute name="body"/>
        </div>
        <div id="divFooter">
            <tiles:insertTemplate template="/WEB-INF/jsp/footer.jsp"/>
        </div>
    </body>
</html>
```

/WEB-INF/jsp/home/index.jsp - exemplo que utilizará o template - repare que o path mantém a convenção do VRaptor3! ^^

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
<tiles:insertTemplate template="/WEB-INF/jsp/template.jsp">

    <tiles:putAttribute name="body">
        olá mundo!
    </tiles:putAttribute>

</tiles:insertTemplate>
```

HomeController.java - e finalmente o controler para redirecionar para index.jsp

```
@Resource

public class HomeController {

    private Result result;

    public HomeController(final Result result) {

        super();
        this.result = result;
    }

    public void index() { }
}
```

Prontinho! Estamos utilizando o tiles apenas para montar o template das páginas, o redirecionamento continua sendo cargo do VRaptor! ^^

Só tem um detalhe que eu gostaria de compartilhar, pois me deu muito trabalho de descobrir: suponha que eu queria estender o template.jsp para incluir um outro body. (tive essa necessidade na seção about do site..)

/WEB-INF/jsp/about/about_base.jsp - base que estenderá a template.jsp para alterar o layout

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>

<tiles:insertTemplate template="/WEB-INF/jsp/template.jsp">

    <tiles:putAttribute name="body">

        <div id="divCenter">
            <tiles:insertAttribute name="content" />
        </div>

        <div id="divRight">
            <p>
            <tiles:insertAttribute name="content_right" />
            </p>
        </div>

    </tiles:putAttribute>

</tiles:insertTemplate>
```

Pronto, agora a minha página do about, estenderá about_base.jsp e não template.jsp, por exemplo:

/WEB-INF/jsp/about/whyUse.jsp - estenderá o about_base.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>

<tiles:insertTemplate template="/WEB-INF/jsp/about/about_base.jsp" flush="true">

    <tiles:putAttribute name="content" cascade="true">
        o conteúdo central!
    </tiles:putAttribute>

    <tiles:putAttribute name="content_right" cascade="true">
                o conteúdo da direita!
    </tiles:putAttribute>

</tiles:insertTemplate>
```

Repare que dessa vez, o insertTemplate possui o atributo flush="true": isso é importante para que a about_base.jsp seja reinderizado primeiro. Note também que os putAttributes possuem o atributo cascade="true": que serve para disponibilizar esses atributos “para os templates de cima”.

Bom, isso foi bem chatinho de descobrir, mas agora a aplicação está rodando bunitinha, sem um xml, com o layout definido nas JSP’s e com o mínimo de interferência no VRaptor3! ^^