**Spring MVC: Requisitos e Fluxo do Processamento da Informação**

O Spring MVC possui uma divisão bem estruturada de suas camadas, tornando o código de uma forma bem organizada.

O framework é compatível com os principais servidores web Java, como o Apache Tomcat, Jboss, BEA Weblogic ou IBM Websphere. Além disso, possui a integração com frameworks para mapeamento do banco de dados como o Hibernate.

Um dos jeitos mais fáceis de se desenvolver um projeto com o Spring MVC é utilizando o Spring Tool Suite que pode ser encontrado no site oficial do framework: <https://spring.io/tools>. Essa ferramenta é um Eclipse com plugins de desenvolvimento do Spring já instalados e configurados, ou seja, nela podemos criar projetos do tipo Spring Project e no assistente de criação, podemos especificar que queremos que o projeto seja do tipo Spring MVC. Com isso não precisamos nos preocupar em baixar as bibliotecas manualmente ou saber quais são as dependências do maven que precisam ser adicionadas.

Mas se vocês quiserem utilizar em um projeto maven do eclipse, ele pode ser baixado adicionando-se a seguinte dependência dentro do pom.xml:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

O fluxo do processo das informações do Spring MVC, segue uma sequência de eventos quando uma requisição é enviada ao framework.

1 - Primeiramente o *DispatcherServlet,* recebe a requisição

O *DispatcherServlet* é um dos principais componentes da estruturação do Spring MVC, pois além de ser um mapeador de requisições, representando um único canal de entrada para todas requisições direcionadas, facilitando o gerenciamento da informação, é responsável por encaminhar para qual *Controller* vai receber e processar a requisição, além de apontar o arquivo de *template* específico no qual será renderizado na camada *View*.

2 - O DispatcherServlet verifica o HandlerMapping e carrega o Controller associado a requisição.

HandlerMapping é uma interface que faz a análise e define um mapeamento da requisição.

3 - O Controller processa a requisição através da chamada aos métodos apropriados do serviço e retorna um objeto ModeAndView para DispatcherServlet.

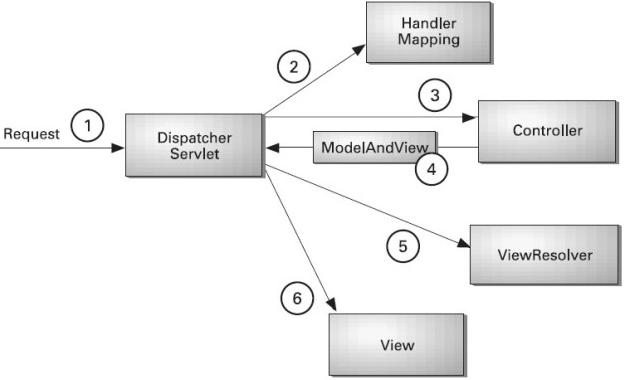
O objeto ModeAndView contém os dados do modelo e o nome da visão.

4 - O DispatcherServlet envia o nome de visão para um ViewResolver para que ele encontre o View que deve ser carregado.

O View Resolver é um gerenciador de visualização, ele procura a página JSP no qual corresponde ao nome da view encaminhada pelo Controller.

5 - Agora o DispatcherServlet passará o objeto modelo para o View para que o resultado seja renderizado.

6 - A View com os dados vindo do modelo, vai renderizar o resultado para o usuário.

****

**Spring MVC: Controllers**

O Controller é uma classe Java que possui os métodos responsáveis por tratar as requisições, e como o próprio nome já diz, faz parte da camada controladora do modelo MVC. Responsável por intermediar as informações junto a View, receber parâmetros e disponibilizar resultados.

O *Controller* suporta os métodos do tipo GET ou POST usado. O método de serviço definirá os dados do modelo com base na lógica de negócios definida e retornará o nome da visualização ao *DispatcherServlet*   
  
Para dizer que a classe será um Controller, basta fazer a anotação **@*Controller*,** assim a classe já passa a servir com a função de um controlador.   
A anotação **@*RequestMapping*** é usada para mapear uma URL para uma classe inteira ou um método de manipulador específico. E para fazer isso, basta passar os atributos como parâmetro de função dentro da anotação.  
O atributo ***value*** indica a URL para a qual o método do manipulador é mapeado e o atributo ***method*** define o método de serviço para manipular a solicitação.

Veja um exemplo simples:

/\*\* Incluir código Spring MVC – C5 \*/

**Spring MVC: Views**

O Spring MVC se destaca por sua separação das tecnologias de visualização. Suporta muitos tipos de visualizações para diferentes tecnologias de apresentação. Estes incluem - JSPs, HTML, PDF, planilhas do Excel, XML, modelos Velocity, XSLT, JSON, Atom e RSS feeds, JasperReports, etc. Mas, mais comumente, são utilizados modelos JSP escritos com JSTL. Veja um exemplo:

//\*\* incluir código Spring MVC – V1 \*/

Aqui **$ {message}** é o atributo que configuramos dentro do Controller. Você pode ter vários atributos para serem exibidos dentro de sua visão.

**Spring MVC: Injeção Dependências**

O Spring MVC foi desenvolvido ao conceito da utilização de Injeção de Dependência, um padrão onde a classe não precisa se preocupar em como conseguir suas dependências, apenas em trabalhar com elas.  
Com isso, ajuda ao desacoplamento do código, tornando mais fácil ao gerenciamento e a realização de testes no sistema.  
E para que o framework identifique os pontos no qual será injetada, é necessário fazer uma anotação na classe com a expressão @Autowired.  
A anotação pode ser utilizada nos em 3 casos:  
- Nas Propriedades;  
- Nos Construtores;   
- Nos Métodos (normalmente, os setters)  
  
Outro requisito para que uma instância possa ser injetada, é transforma-lo em uma Bean Spring.   
No qual é necessário fazer a anotação @Component ou com qualquer uma de suas especializações:  
  
@Component: Esta anotação faz com que o bean registrado no Spring possa ser utilizado em qualquer bean, seja ele um serviço, um DAO, um controller, etc.  
@Service: Anotação no qual diz que a bean faz parte da camada de serviço.   
@Repository: Anotação no qual diz que a bean faz parte da camada de persistência.

//\*\* incluir código Spring MVC – Di1 \*/

**Spring MVC: Interceptadores**

E para entender o interceptador, vamos dar um passo para trás e ver o HandlerMapping . Isso mapeia um método para uma URL, para que o DispatcherServlet possa invocá-lo ao processar uma solicitação.

E o DispatcherServlet usa o HandlerAdapter para invocar o método.

Agora que entendemos o contexto geral, **é aqui que entra o interceptador**. Usaremos o  HandlerInterceptor para executar ações antes do manuseio, após o manuseio ou após a conclusão (quando a exibição for renderizada) de uma solicitação.

O interceptor pode ser usado para interesses transversais e para evitar códigos manipuladores repetitivos, como: registro em log, alteração de parâmetros usados ​​globalmente no modelo Spring etc.

Os interceptores que trabalham com o HandlerMapping na estrutura devem implementar a interface HandlerInterceptor.

Esta interface contém três métodos principais:

* prehandle () - chamado antes do manipulador real ser executado, mas a visualização ainda não foi gerada
* postHandle () - chamado depois que o manipulador é executado
* afterCompletion () - chamado depois que a solicitação completa foi concluída e a visualização foi gerada

Esses três métodos fornecem flexibilidade para todos os tipos de pré e pós-processamento.

E uma nota rápida - a principal diferença entre HandlerInterceptor e HandlerInterceptorAdapter é que, no primeiro, precisamos substituir todos os três métodos: preHandle () , postHandle () e afterCompletion () , enquanto no segundo, podemos implementar apenas os métodos necessários.

/\*\* Inserir código Spring MVC Interceptor – I2 \*/

**Spring MVC: Validadores**

A validação da entrada recebida do usuário para manter a integridade dos dados é uma parte importante da lógica do aplicativo. A validação de dados pode ocorrer em diferentes camadas.

O Spring MVC no qual é baseado na plataforma JAVA EE 6, aproveita um dos recursos disponíveis para fazer a validação. O modelo Bean Validation, no qual é suportado por restrições na forma de anotações colocadas em um campo, método ou classe de um componente JavaBeans, como um bean gerenciado e pode ser utilizado em qualquer camada da aplicação.

Com o Bean Validation declaramos através de anotações as regras de validação dentro do nosso modelo:

/\*\* Inserir código Spring MVC Validator – V1 \*/

Com a anotação do Bean Validation na camada de controller. É necessário avisar o Spring MVC que queremos executar a validação. Isso é feito pela anotação Valid que devemos usar na antes do parâmetro da ação:

/\*\* Inserir código Spring MVC Validator – V2 \*/

O Spring MVC pode armazenar o resultado dos erros de validação em um objeto do tipo BindingResult. Este objeto BindingResult se torna um parâmetro da ação. No qual pode ser utilizado para que em vez de ser lançado uma exceção, seja redirecionado para outra página.

/\*\* Inserir código Spring MVC Validator – V3 \*/

/\*\* Inserir código Spring MVC Validator – V4 \*/

Para a exibição das mensagens de validação na camada da View, é utilizado uma tag especial que o Spring MVC oferece. A tag se chama **form:errors**:

<form:errors path="tarefa.descricao" />

O atributo path indica com que atributo essa mensagem está relacionada.

/\*\* Inserir código Spring MVC Validator – V5 \*/

**Spring MVC: Plugins**

O Spring MVC possui diversos módulos disponíveis para acoplamento no seu projeto, para facilitar ainda mais do desenvolvimento para serviços específicos.

• Spring Security útil para inclusão de funcionalidades de autenticação e autorização.

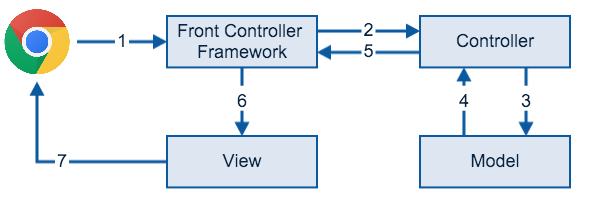
• Spring Data para aplicações que usam novas tecnologias de armazenamento de dados e serviços na nuvem.

• Spring Social para fácil integração com redes sociais.

• Além de outros. Disponível no site oficial do framework <http://spring.io/projects>

**Modelo MVC**

Segue abaixo a imagem de uma representação do fluxo do processo de uma requisição do ponto de vista do padrão MVC

[](https://s3.amazonaws.com/algaworks-blog/wp-content/uploads/Fluxo-do-Spring-MVC.png)

Os detalhes dos passos citado na imagem acima, segue da sequinte forma:  
  
1. Fazemos uma requisição HTTP através de uma URL no navegador, no qual é enviado para um servidor web com Spring MVC.  
2. O framework então através do seu *controller* (controlador), faz a pesquisa para localizar qual a classe é responsável por tratar a requisição.  
3. O *controller* encaminha os dados para o *model (modelo),* seguindoas boas práticas, camada responsável por executar as regras de negócios, por exemplo: validações, cálculos e acesso ao banco de dados.  
4. Após realizar as operações, o *model* retorna o resultado ao *controller.*  
5. O controller, então devolve o endereçamento da *view* (visão), junto aos dados que serão renderizados junto a página.  
6. O framework localiza a view que será processado os dados, renderizando o resultado em uma página web.   
7. Finalizando, a página web é retornando ao navegador do usuário.

**VRaptor: Requisitos e Fluxo do Processamento da Informação**

Pré requisito JDK 7 e o CDI 1.1 ou superiores, criando projetos Download disponível pelo <https://bintray.com/caelum/VRaptor4/br.com.caelum.vraptor/> possui 2 opções:

- Vraptor Blank Project é um projeto preparado com mínimo necessário para rodar o VRaptor, usando o Maven para gerenciar as dependências

- Na página de download possui também o zip de distribuição, que contém a distribuição completa da última versão do VRaptor. Nesse zip podemos encontrar o jar do VRaptor, suas dependências (pasta lib), seu javadoc (pasta apidoc) e código fonte (pasta src). Assim já é possível linkar esses artefatos na sua IDE (Eclipse, Netbeans, etc.) e facilitar o desenvolvimento.

- Servidores suportados Wildfly 8, Tomcat 7 e Jetty 8.

- Para manipulação dos dados junto ao banco de dados, recomendável a utilização do Hibernate.

O Hibernate é um framework **ORM - Object Relational Mapping**. É uma ferramenta que nos ajuda a persistir objetos Java em um banco de dados relacional. O trabalho do desenvolvedor é definir como os objetos são mapeados nas tabelas do banco e o Hibernate faz todo o acesso ao banco, gerando inclusive os comandos SQL necessário.

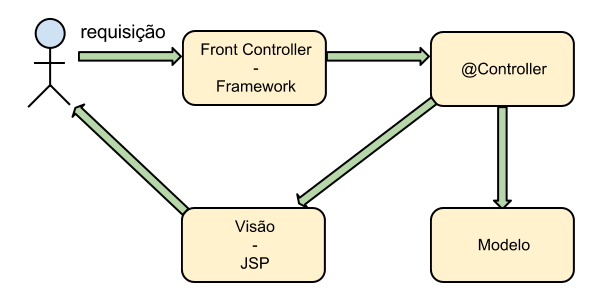
VRaptor trabalha na camada de Controller, ele é quem controla as entradas e dispara as requisições internas para os controllers e suas views.

Quando uma requisição chega, ela é prontamente atendida pelo VRaptor.

O framework então decide qual action chamar.

A action executa, e ao final, diz ao framework qual JSP exibir.

O VRaptor por fim, pega a JSP, a processa, e envia para o usuário final, finalizando a requisição.



No exemplo anterior, os dados que precisam ser disponbilizados para a Visão fazer a exibição são “empurrados” do @Controller. Quando isso acontece, dizemos que estamos usando um estilo de MVC chamado MVC Push, ou seja, que empurra os dados para a Visão utilizar. Atualmente, frameworks que seguem essa estratégia são conhecidos como Action Based (Baseado em Ações).

A convenção de URL do VRaptor é controllerName / methodName, portanto, o método list será acessível por user / list. Observe que o sufixo do controlador não é considerado na rota.

Os despachos da JSP seguem outra convenção útil. É bastante semelhante à convenção de URL: após a execução do método do controlador, o VRaptor irá procurar o JSP no

WEB-INF / jsp / controllerName / methodName.jsp.

Em nosso exemplo, será

WEB-INF / jsp / user / list.jsp.

Todos os métodos públicos do seu controlador serão mapeados seguindo estas convenções e responderão a solicitações independentemente do verbo HTTP usado.

Você também pode limitar o acesso a um método controlador de acordo com o verbo escolhido, usando as anotações @Get, @Post, @Put ou @Delete.

Se você optar por não usar a convenção de URLs, também poderá adicionar um parâmetro String a essas anotações, alterando a rota da URL que os alcançará e, se quiser personalizar o URL de um método irrestrito de verbo HTTP, você poderá usar o @ Anotação de caminho.

**VRaptor: Controllers**

Tudo que se precisa fazer para criar um controller do VRaptor é adicionar a anotação @Controller. A partir daí o framework já utiliza suas convenções de URLs e JSPs, exigindo o mínimo de configurações. Veja um exemplo:

/\*\* Inserir código VRaptor Controller – C1 \*/

A convenção de URLs do VRaptor é *nomeDoController/nomeDoMetodo*; portanto o método lista() seria mapeado para a URL */usuario/lista*. Repare que o sufixo *Controller* não é considerado na rota.

Outra convenção importante é da página JSP pra a qual o VRaptor vai fazer o dispatch. Ela é bem parecida com a convenção de URLs.O VRaptor procura a JSP no diretório:

WEB-INF/jsp/nomeDoController/nomeDoMetodo.jsp,

que neste caso será:

WEB-INF/jsp/usuario/lista.jsp.

Todos os métodos públicos dos controllers serão mapeados seguindo essas convenções, e passam a atender requests independente do verbo HTTP utilizado.

Pode-se explicitamente declarar qual tipo de requisição cada método vai atender, por exemplo utilizando a anotação @Get. Há ainda as anotações @Post, @Put e @Delete. Se preferir não utilizar a convenção de URLs, você pode passar a URL como *value* dessas anotações de verbo, ou utilizar a anotação @Path se o verbo for indiferente:

Utilizado como classe que recebe os dados da view e coordena as lógicas de negócio

**Annotation**

@Resource – São recursos a serem disponibilizados para acesso pelos nossos clientes

@Component – Instancias de classes que seu projeto precisa para executar tarefas. Exemplos clássicos de uso de componentes seriam os DAOs

@Path – Informa a url que o método de um recurso atenderá

@Post – Informa que o método atendera apenas requisição do tipo Post

@Get – Informa que o método atendera apenas requisição do tipo Get

Controllers

São classes Java

Recebem requisições do usuário num determinado contexto

Possuem métodos que executam tarefas específicas

Tem que ser anotados com @Resource para que seus métodos públicos sejam visíveis para os clientes Podem receber dependências através do construtor ou através dos parâmetros dos métodos

**CONVENÇÕES**

- Classes controladoras devem ter seu nome terminando com “Controller”, e anotadas com @Resource

- Observação: Não tem problema em ter Controllers em outros pacotes, desde que não existam classes com o mesmo nome.

- Arquivos JSP referentes a determinado controlador devem estar em uma pasta com o seu nome (estilo lowerCamelCase, e excluindo a palavra “Controller”), dentro da pasta /WEB-INF/jsp

- O nome do arquivo JSP precisa ser igual o nome do método no controlador

- A URL de acesso fica no formato: domínio/contexto/controlador/método

Exemplo: localhost:8080/meuProjeto/produto/cadastrar

- Podemos alterar a URI padrão gerada para determinado método com a anotação @Path

Exemplo:

Public class ProdutoController{

@Path(“/produto/novo”)

Public void adiciona(){...}

- Nos controladores, métodos públicos e sem nenhuma anotação são considerados métodos Get

- Podemos receber parâmetros via URL e injeta-los nos objetos recebidos no método

@Get

@Path(“/produto/{produto.id}”)

Public Produto visualiza(Produto produto) {...}

/\*\* Inserir código VRaptor Controller – C2 \*/

**VRaptor: Views**

O VRaptor possui grande flexibilidade na escolha de tecnologias da camada de visão, devido ao baixo acoplamento com o controlador proporcionado pelo modelo MVC baseado em ações. É possível utilizar vários templates para geração de páginas dinâmicas, como por exemplo JavaServer Pages (JSP), Velocity, Freemarker, entre outros. Entretanto, os

componentes visuais devem ser desenvolvidos manualmente, utilizando bibliotecas externas como o JQuery UI, Bootstrap, ExtJS, AngularJS e etc.

A camada de visão não precisa ser, necessariamente, gerada dinamicamente no

servidor através de uma ferramenta de template. No caso do cliente ser outra aplicação ou simplesmente uma página HTML estática, a comunicação pode ser feita através da

serialização de objetos em determinado formato, como XML ou JSON. Este tipo de solução é comum ao disponibilizar serviços web para integração entre sistemas.

Compartilhando resultado na views

Para registrar objetos a serem acessados na view, usamos o método include:

/\*\* Inserir código VRaptor Views – V1 \*/

Agora as variáveis mensagem e cliente estão disponíveis para uso em seu template engine. É possível registrar o objeto por meio da invocação do método include com um único argumento:

/\*\* Inserir código VRaptor Views – V2 \*/

Nesse caso, a primeira invocação registra a chave string e a segunda, a chave cliente. Você pode alterar o comportamento de convenção de chaves no seu próprio TypeNameExtractor.

Por padrão, para renderizar suas views, o VRaptor segue a convenção:

/\*\* Inserir código VRaptor Views – V3 \*/

Este método acima renderizará a view /WEB-INF/jsp/clients/list.jsp. No entanto, nem sempre queremos esse comportamento, e precisamos usar algum template engine, como por exemplo, Freemarker ou Velocity, e precisamos mudar essa convenção. Um jeito fácil de mudar essa convenção é estendendo a classe DefaultPathResolver:

/\*\* Inserir código VRaptor Views – V4 \*/

Desse jeito, a lógica irá renderizar a view /WEB-INF/freemarker/clients/list.ftl. Se, ainda assim, isso não for o suficiente, você pode implementar a interface PathResolver e fazer qualquer convenção que você queira, não esquecendo de anotar a classe com @Specializes.

Se você quiser mudar a view de alguma lógica específica, você pode usar o objeto Result:

/\*\* Inserir código VRaptor Views – V5 \*/

Por padrão, existem estes tipos de views implementadas:

O VRaptor disponibiliza um objeto para trabalhar com alguns recursos relacionados a View, este objeto chamado de Result, pode ser injetado através do construtor

- O Result pode redirecionar o fluxo para outra lógica de outro controlador

Result.redirect()

- O Result pode modificar a view padrão, retornando JSON, XML, Status HTTP, ao invés de JSP.

Result.use(Results.json()).serialize();

- O Result pode adicionar objetos no request, tornando-os acessíveis na JSP:

Result.include(“mensagem”, “Senha alterado com sucesso”)

- O Result pode redirecionar o fluxo caso ocorra uma Exception:

Result.on(GenericAccessDeniedException.class).redirectTo(AccessDeniedController.class).index();

- Conversão automática de tipos;

Por padrão, são páginas JSP convencionais

Ao terminar a execução do método no Controller, o VRaptor vai fazer o dispatch da requisição para o jsp adequado.

A convenção para a view padrão é: /WEBINF/jsp//.jsp

É possível alterar a convenção padrão, até mesmo processar tipos de resposta diferentes de html, como xml e json

**Components**

Classes que, normalmente, pertencem à camada de modelo

Devem ser anotadas com @Component pra que o VRaptor possa injetá-las onde for necessário

Podem receber suas dependências através do construtor

Possuem tempo de vida diferenciado dos controllers

**VRaptor: Injeção de Dependências**

A Inversão de Controle é um padrão de desenvolvimento onde se inverte o fluxo normal de programação, delegando a responsabilidade sobre algumas partes do código para alguns frameworks ao invés de criarmos toda a sequencia

Só para lembrar: Inversão de controle é um padrão de desenvolvimento, e Injeção de Dependências é uma maneira de resolver/implementar a IoC.

Todos os componentes do VRaptor são gerenciados pelo CDI (Context Dependency Injection) do Java EE 7. Desta forma todas as funcionalidades presentes no CDI estão presentes no VRaptor.

o VRaptor também utiliza um framework de injeção de dependências para controlar o

ciclo de vida de seus componentes.

, que é a especificação padrão de um framework de injeção de dependências para a plataforma Java EE.

O VRaptor tira proveito das funcionalidades do CDI para manter boas práticas de

desenvolvimento, torna o framework totalmente extensível, além da possibilidade de integração com os recursos nativos do servidor de aplicação e com as demais especificações

contidas na plataforma Java EE.

O VRaptor utiliza o CDI para controlar o que é necessário para instanciar cada um de seus componentes e recursos. Sendo assim, os dois exemplos anteriores permitem que quaisquer um dos seus recursos ou componentes recebam um ClienteDao em seu construtor, para isso o CDI nos obriga a termos o construtor padrão, por exemplo:

/\*\* Inserir código VRaptor DI – D1 \*/

Nosso fluxo principal agora é tratado com [eventos do CDI](http://blog.caelum.com.br/diminua-suas-dependencias-com-os-eventos-do-cdi/), assim nosso código ficou ainda mais desacoplado e extensível. [Nessa página da documentação](http://caelum.github.io/vraptor4/pt/docs/eventos/) exibimos um mapa dos eventos e seus observers, assim como uma explicação simples sobre o que cada um deles representa.

Você pode facilmente observar os eventos do VRaptor, para isso basta escrever um observer simples como esse:

|  |
| --- |
| import javax.enterprise.event.Observes;  import br.com.caelum.vraptor.events.ControllerNotFound;    public class ControllerNotFoundHandler {      public void metodo(@Observes ControllerNotFound evento) {      // alguma ação para quando o controller não for encontrado    }  } |

**VRaptor: Interceptadores**

Um dos principais componentes que o VRaptor oferece é o interceptador, que é análogo ao clássico Filter da Servlet, porém integrado ao contexto de injeção de dependências

existem tarefas ou funcionalidades que impactam boa parte da aplicação, como

por exemplo o controle de acessos, controle de transações, logs de erros e etc. Geralmente, quase todas as funcionalidades da aplicação passam por esse tipo de controle, que deve ser implementado em um único ponto do código, facilitando a manutenibilidade. O interceptador permite que o desenvolvedor registre funções de callback antes e depois da execução de cada Controller.

Se você precisa de ordenação na execução de seus eventos, considere utilizar Interceptors. O VRaptor 4 possui um novo modelo baseado em anotações! Veja como pode escrever seu interceptor:

|  |
| --- |
| @Intercepts  public class ApplicationInterceptor {        @Accepts      public boolean accepts(ControllerMethod method) {          return method.containsAnnotation(Audit.class);      }        @BeforeCall      public void before() {          // código a ser executado antes da lógica      }        @AfterCall      public void after() {          // código a ser executado depois da lógica      }        @AroundCall      public void intercept(SimpleInterceptorStack stack) {          // código a ser executado antes da lógica          stack.next(); // continua a execução          // código a ser executado depois da lógica      }  } |

Um interceptor sem o método anotado com @Accepts é global, ou seja, vai interceptar todas as requisições. Outra novidade dos interceptors é que você pode utilizar os aceptors customizados como o @AcceptsWithPackage e @AcceptsWithAnnotations:

|  |
| --- |
| @Interceptor  @AcceptsWithAnnotations(Audit.class)  public class AuditInterceptor { ... } |

**VRaptor - Validadores**

O principal método de validação do VRaptor é baseado na especificação Bean Validation, presente na plataforma Java EE. Entretanto, é possível utilizar o validador próprio do

framework. (VRAPTOR, 201 5).

VRaptor possui uma API fluente de validações, dando suporte para internacionalização de forma simples. Permite ao desenvolvedor tomar determinada ação quando uma restrição de validação é violada, como por exemplo redirecionar para outro método de outro Controller ou simplesmente retornar um status de erro.

- Classe Validator deve ser injetada

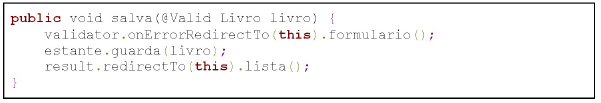
- Possui o Estilo clássico, estilo fluente, Bean Validation

Validadores clÃ¡ssicos
  Validadores Fluente
 

Validadores Bean Validation
 

O quadro 7 apresenta um método com regra de

redirecionamento no caso de uma restrição de validação.

****

**VRaptor: Plugins**

Ao atingir certo nível de maturidade e conquistar espaço na comunidade de desenvolvimento, diversos plugins surgiram através de iniciativas de terceiros.

VRaptor facilita a criação de componentes reusáveis que tem por objetivo resolver

problemas em comum e podem ser facilmente adicionados em qualquer aplicação. Existe um catálogo de plugins disponíveis para o VRaptor, alguns criados pela Caelum e

outros pela própria comunidade.

Dentre diversos outros aqui não listados, os seguintes plugins já possuem uma versão compativel com essa nova versão do framework:

[vraptor-time-converters](https://github.com/caelum/vraptor-time-converters) – trabalhar com date time e java time  
[vraptor-simplemail](https://github.com/caelum/vraptor-simplemail) – já conhecido plugin de envio de emails  
[vraptor-errorcontrol](https://github.com/caelum/vraptor-error-control) – controle de erros efetivo com envio de e-mails  
[vraptor-quartzjob](https://github.com/caelum/vraptor-quartzjob) – schedule Quartz para agendamento de tarefas  
[vraptor-freemarker](https://github.com/caelum/vraptor-freemarker) – trabalhar com templates do freemarker  
[vraptor-jpa](https://github.com/caelum/vraptor-jpa) e vraptor-hibernate – produtores e controle de transação  
[vraptor-brutauth](https://github.com/leonardowolter/vraptor-brutauth), [vraptor-shiro](https://github.com/dipold/vraptor-shiro" \t "_blank) e [vraptor-authz](https://github.com/Turini/vraptor-authz/" \t "_blank) – autent. e autorização  
[vraptor-dash](https://github.com/caelum/vraptor-dash) – dashboard com diversar ferramentas pro seu projeto

Esses e muitos outros recursos já estão implementados e distribuídos como plugins do VRaptor4. Os interceptores e produtores mencionados acima foram colocados nos plug [-](https://github.com/caelum/vraptor-hibernate) ins [vraptor-jpa](https://github.com/caelum/vraptor-jpa/) e [vraptor-hibernate](https://github.com/caelum/vraptor-hibernate) , e tudo o que precisamos fazer para usá-los é adicionar seu jar (ou até mesmo configurar seu gerenciador de dependências favorito) em nosso projeto. Nenhuma configuração extra é necessária!

Enquanto o plug-in tiver um arquivo beans.xml, o CDI gerenciará suas classes e as disponibilizará para serem injetadas no VRaptor. Como criar extensões é muito fácil, a participação da comunidade tem sido ótima e muitos plugins foram criados. Alguns deles podem ser encontrados na documentação da estrutura [nesta página](http://www.vraptor.org/en/docs/plugins/) .

Referencias

● http://www.vraptor.org/pt/

● http://www.casadocodigo.com.

br/products/livro-vraptor

● http://getbootstrap.com/

● http://api.jquery.com/

● http://api.jqueryui.com/

<http://respostas.guj.com.br/tag/vraptor>

Comunidade:

<http://github.com/caelum/vraptor>

[caelum-vraptor@googlegroups.com](mailto:caelum-vraptor@googlegroups.com)

http://vraptor.