**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.

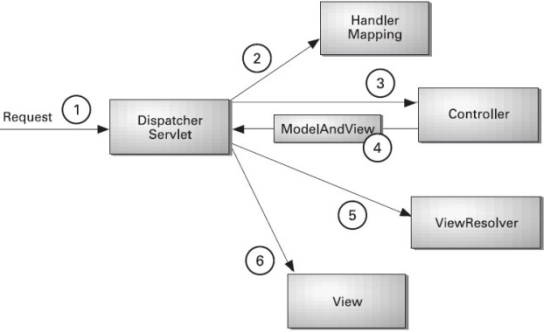
O objeto ModeAndView contém os dados do modelo e o nome da visão.

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

OView Resolver é um gerenciador de visualização, ele procura a página JSP no qual corresponde ao nome da viewencaminhada 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,vairenderizar 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 servircom 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 umaBean Spring.   
No qual é necessáriofazer 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 beanfaz 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 BeanValidation, 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 BeanValidation declaramos através de anotações as regras de validação dentro do nosso modelo:

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

Com a anotação do BeanValidation 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 umatag especial que o Spring MVC oferece. Atag se chama **form:errors**:

<form:errorspath="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.

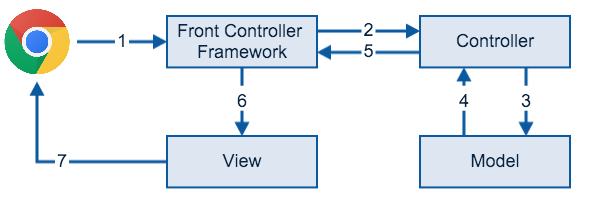
• Spring Data para aplicações que usam novas tecnologias de armazenamento de dados e serviços na nuvem.

• 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**

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

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:

VraptorBlank 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.

Os Servidores suportados e já testados pela própria framework, são Wildfly 8, Tomcat 7 e Jetty 8.

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

Assim como o Spring MVC, o VRaptor é baseado o seu fluxo de processamento da informação em Action Based, que recebe diretamente as requisições HTTP. Tornando o modelo flexível, deixando a livre opção de escolha do tipo de view para gerar uma requisição HTTP compatível.

O VRaptor possui tem o benefício de encapsular as principais classes de Servlets, por exempo as classes HttpServletRequest, HttpServletResponse e Session, obtendo assim o ganho de poder tratar a regra de negócio por Controllers.

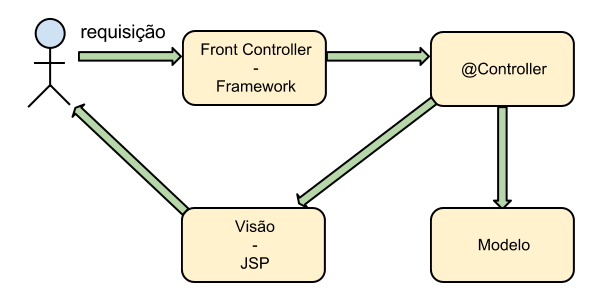
O VRaptor foca em simplicidade, baseado em **convenção sobre configuração,** com intuito de facilitar a padronização do código e evita as diversas configurações em arquivos XML vistas em outros frameworks.

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

- O framework então através da anotação do Controller, 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.



**Segue algumas das anotações utilizadas para o gerenciamento da informação do VRaptor:**

@Controller – Anotação responsável por identificar os elementos que estarão disponíveis como controladores de requisição

@Component – Anotação para indicar que aquela classe usará o padrão de Injeção de Dependências para executar tarefas.

@Path – Anotação utilizado para informar a URL que será atendido a requisição

**VRaptor: Controllers**

Para criar o Controller no VRaptor, que são classes que vão executar as tarefas de requisição, basta apenas adicionar a anotação @Controller na classe desejada, para que seus métodos publico estejam disponíveis. A partir daí o framework necessita que sejam seguidas algumas regras de convenções de URLs e JSPs, para funcionar corretamente.

As classes controladoras devem possuir o nome com a terminação *Controller*, e anotadas com @*Controller*.

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

A URL de acesso deve seguir o seguinte formato: domínio/contexto/controlador/método

Exemplo: localhost:8080/meuProjeto/produto/cadastrar

/\*\* Inserir código VRaptorController– C1 \*/

Seguindo corretamente essas convenções, todos os métodos públicos dos *controllers* serão mapeados, passando a executar as requisições.

Outra vantagem que o VRaptor utiliza do conceito de anotações em classes, é a possibilidade de declarar o tipo de requisição que será executado. Bastando apenas, assim no *controller*, anotar a classe com os seguintes tipos:

@Post – Anotação para dizer que o método atendera apenas requisição do tipo Post

@Get – Anotação para dizer que o método atendera apenas requisição do tipo Get

@Put – Anotação utilizado para realizar alterações em parte ou no objeto inteira

@Delete - Anotação utilizado para realizar a exclusão de algum objeto ou elemento

/\*\* Inserir código VRaptorController– C2 \*/

**VRaptor: Views**

O VRaptor tem a característica de possuir baixo acoplamento da camada visão com seu controlador, devido ao modelo baseado no MVC de ações. Ele torna-se flexível a escolha da tecnologia da visualização da interface.

Pode ser utilizado diversos templates, por exemplo: JSP, Velocity ou Freemaker para criação de paginas dinâmicas, entretanto as ações e os elementos visuais ainda sim, devem ser criados manualmente ou utilizando bibliotecas externas como por exemplo: Bootstrap, JQuery, AngularJS, etc.

A convenção padrão utilizada na view, é no qual os arquivos JSP devem estar dentro da pasta /WEB-INF/jsp com o nome referente ao controlador correspondente, excluindo a terminação Controller e seguindo o estilo lowerCamelCase,

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. Este tipo de solução é comum ao disponibilizar serviços web para integração entre sistemas.

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).principal();

- Conversão automática de tipos;

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

/\*\* Inserir código VRaptorViews– V1 \*/

/\*\* Inserir código VRaptorViews– V2 \*/

/\*\* Inserir código VRaptorViews– V3 \*/

**VRaptor: Injeção de Dependências**

O VRaptor se beneficia de todas funcionalidades e as boas práticas fornecidas pelo CDI do Java EE 7, pois todos os componentes, que são instancias de classes necessários para execução de tarefas, o ciclo de vida de seus componentes e a possibilidade de armazenar o estado de interação do usuário, são todos gerenciados através dele.

Isso torna o framework mais desacoplado e extensível, além da integração com recursos nativos do servidor de aplicação com as demais especificações contidas na plataforma.

Para implementação, é necessário declarar um construtor padrão para que o CDI possa gerenciar a classe, ao componente gerenciável fazer a anotação @Named e para identificar o ponto no qual será injetavel com a anotação @Inject.

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

Um exemplo do VRaptor gerenciando o ciclo de um componente através do escopo:

/\*\* Inserir código VRaptor DI – D2 \*/

**VRaptor - Validadores**

VRaptor utiliza como principal método de validação a especificação do BeanValidation, fornecidos também pelo Java EE 7, através dele podemos validar todos os modelos baseado em anotações.

Entretanto, pode ser utilizado os métodos de validação do próprio framework, através do método add() para retorno de uma mensagem simples ou internacionalizada, e o método addIf() para exibir a mensagem na condição do método ser verdadeiro ou ensure() para exibição d mensagem sob a condição falsa, todos métodos pertencentes da classe Validator do VRaptor.

Exemplo de validação com validator do VRaptor:

Exemplo da validação com BeanValidation com regra de redirecionamento no caso de uma restrição de validação.

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

**VRaptor: Plugins**

O VRaptor posssui diversos plugins disponibilizados em seu site oficial, no qual muitos foram criados pela Caelum ou pela própria comunidade, pois muitos desses plugins foram criados através de terceiros, por desenvolvedores que possui um nível mais avançado e reconhecimento na comunidade, pois o framework tem essa facilidade de criar componentes reusáveis e de fácil aplicação, com intuito de sempre tentar resolver um problema em comum.

Para utilizar esses plugins, é necessário apenas adicionar o arquivo jar no seu projeto, pois enquanto o plugin possuir o arquivo bean.xml, o CDI vai fazer o gerenciamento e a disponibilização das classes a serem injetadas.

Segue alguns plugins que já possuem uma versão compatível com versão do framework:  
[vraptor-time-converters](https://github.com/caelum/vraptor-time-converters) – para converter com data e hora   
[vraptor-simplemail](https://github.com/caelum/vraptor-simplemail) –plugin para facilitar o envio de emails  
[vraptor-quartzjob](https://github.com/caelum/vraptor-quartzjob) – realizar agendamento de tarefas  
[vraptor-jpa](https://github.com/caelum/vraptor-jpa) e vraptor-hibernate – produtores e controle de transação

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>

<https://docs.oracle.com/javaee/7/tutorial/cdi-basic008.htm>

Comunidade:

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

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

<http://vraptor>.

**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  publicclassApplicationInterceptor {        @Accepts      publicbooleanaccepts(ControllerMethodmethod) {          returnmethod.containsAnnotation(Audit.class);      }        @BeforeCall      publicvoidbefore() {          // código a ser executado antes da lógica      }        @AfterCall      publicvoidafter() {          // código a ser executado depois da lógica      }        @AroundCall      publicvoidintercept(SimpleInterceptorStackstack) {          // 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)  publicclassAuditInterceptor { ... } |