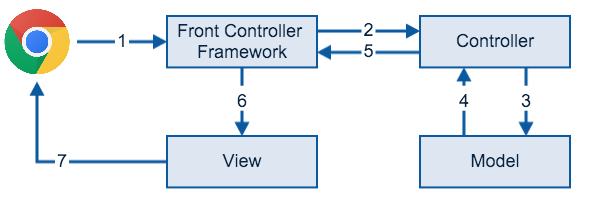
**Modelo MVC**

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.

**Spring MVC: Arquitetura e Requisitos**

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 nesse link: <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>

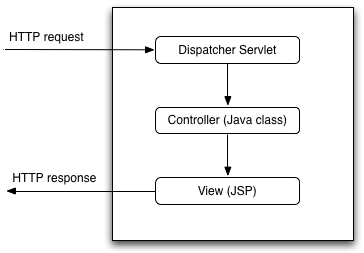
<artifactId>spring-webmvc</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

O Spring MVC ajuda a construir aplicações web flexíveis e com baixo acoplamento. O padrão de design Modelo-visão-controlador ajuda na separação da lógica de negócio, lógica de apresentação e lógica de navegação. Os Modelos são responsáveis por encapsular os dados da aplicação. As Visões renderizam as repostas ao usuário com a ajuda do objeto modelo. Os Controladores são responsáveis por receber as requisições do usuário e executar os serviços.

A figura abaixo mostra o fluxo de uma requisição no Framework Spring MVC:



Quando uma requisição é enviada ao Framework Spring MVC a seguinte sequência de eventos acontece.

* O DispatcherServlet recebe a requisição em primeiro lugar.
* O DispatcherServlet consulta o HandlerMapping e invoca o Controller associado a requisição.
* 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.
* O DispatcherServlet envia o nome de visão para um ViewResolver para que ele encontre o View que deve ser invocado.
* Agora o DispatcherServlet passará o objeto modelo para o View para que o resultado seja renderizado.
* O View com a ajuda dos dados do modelo irá renderizar o resultado para o usuário.

O funcionamento básico do Spring MVC se dá através da classe **DispatcherServlet**, que é um Front Controller e estende **HttpServlet**. Ela tem como funções principais direcionar as requisições para outros controladores, tratá-las e repassar as respostas para o usuário.

Após o recebimento da solicitação HTTP, o **DispatcherServlet** consulta o **HandlerMapping**, que redireciona a solicitação para outro controlador de acordo com suas configurações de comportamento.

**Figura 1**. Funcionamento do Spring MVC.

**Handler Mappings**

Você pode mapear manipuladores para solicitações HTTP de entrada no contexto do aplicativo Spring

Esses manipuladores geralmente são controladores mapeados para URLs parciais ou completos

de solicitações recebidas.Os mapeamentos do manipulador também podem conter interceptores opcionais,

que são invocados antes e depois do manipulador. Este é um conceito poderoso. eu vou

demonstrar um exemplo disso mais adiante neste capítulo, quando usamos um interceptador da Web

para autenticação e feche nossa sessão do Hibernate para a solicitação HTTP fornecida.

**Spring MVC: Controllers e Convenções**

Na prática, o *controller* é a classe Java com os métodos que tratam essas requisições. Portanto, tem acesso a toda informação relacionada a ela como parâmetros da URL, dados submetidos através de um formulário, cabeçalhos HTTP, etc.

Veja um exemplo simples:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Controller  public class HomeController {      @RequestMapping("/")    public String index() {      return "index.jsp";    }  } |

* **6**

Marcar como lidoAnotarImprimir

### Afinal, o que é Spring MVC?

Atualmente no desenvolvimento de software, o grande “bam bam bam” dos modelos de desenvolvimento é o MVC (“Model-View-Controller”), que foi descrito pela primeira vez por volta de 1970 por Trygve Reenskaug que trabalhavam com o Smalltalk (Linguagem de programação orientada a objeto). Foi criado para fugir da antiga bagunça, onde o código se misturava entre a página e a persistência. Esse modelo facilita a manutenabilidade do código, pois a mudança na view (página) não afeta o “Controller” ou o “model”, e vice versa.

**O Spring, como não poderia ficar de fora**, foi criado baseado no[**Spring MVC**](http://www.devmedia.com.br/introducao-ao-spring-mvc/3378), onde temos o @Controller (que abordaremos mais a fundo nesse artigo), @Service (facilitador para acessar os Models a partir de um framework de persistência de sua escolha ou JDBC puro) e as Views. As nossas views são [**JSP**](https://www.devmedia.com.br/introducao-ao-java-server-pages-jsp/25602), pois não é apenas isso que o framework possui.

Agora, vamos nos aprofundar nos Controllers do Spring e entender o que é possível criar com eles.

Relacionado: [**O Que é o Spring Framework?**](https://www.devmedia.com.br/curso/o-que-e-spring/2094)

### Requisitos

Um Dinamic Web Project configurado com as bibliotecas do **Spring e rodando corretamente**.

Observação: Ver referência no fim do artigo para acessar artigo com exemplos de configuração - usaremos o mesmo projeto contido no artigo.

Um Controller é responsável tanto por receber requisições como por enviar a resposta ao usuário, algo bem parecido com o Servlet do JSP. Porém, é feito de forma mais elegante e fácil. O Controller se responsabiliza por informar a View, os atributos que serão visíveis para a mesma e também por receber parâmetros vindos da View. E, por último, responder ao usuário o que foi requisitado. Veja uma exemplo na **Listagem 1**.

**Listagem 1:** Criando um controller

|  |
| --- |
| @Controller //Define que minha classe será um controller  public class HomeController {    } |

Veja como é simples criar um Controller, mas veja que este não possui nenhum “mapping” atrelado a ele. Então criemos uma view (.jsp) chamada “home.jsp” dentro da pasta “/WEB-INF/views” e criaremos o mapping “/home” para exibir a view criada. Veja a **Listagem 2**.

**Listagem 2:** Criando um RequestMapping para HomeController

|  |
| --- |
| @Controller //Define que minha classe será um controller  public class HomeController {        @RequestMapping("/home") //Define a url que quando for requisitada chamara o metodo      public ModelAndView home(){           //Retorna a view que deve ser chamada, no caso home (home.jsp) aqui o .jsp é omitido          return new ModelAndView("home");      }    } |

Agora sabemos como fazer uma requisição e retornar uma pagina, mas vamos supor que minha sessão home possuam várias páginas dentro dela, por exemplo /home/principal e /home/secundaria. Para isso, vamos fazer uma pequena alteração no nosso “HomeController”, conforme a **Listagem 3**.

**Listagem 3:** Modificando o HomeController

|  |
| --- |
| @Controller //Define que minha classe será um controller  @RequestMapping("/home") //Define que qualquer ação desse controler deve preceder /home  public class HomeController {        //Define a url que quando for requisitada chamara o metodo no caso /home/principal ou /home/ ou /home      //Note que não é obrigado apenas uma url, pode-se mapear varias para o mesmo metodo      @RequestMapping( value = { "/" , "" ,"/principal" })      public ModelAndView homePrincipal(){           //Retorna a view que deve ser chamada, no caso principal (principal.jsp) que esta dentro da pasta /home          return new ModelAndView("home/principal");      }        //Define a url que quando for requisitada chamara o metodo      @RequestMapping( "/secundaria" )      public ModelAndView homeSecundaria(){           //Retorna a view que deve ser chamada, no caso secundaria (secundaria.jsp) que esta dentro da pasta /home          return new ModelAndView("home/secundaria");      }    } |

Observação: Não necessariamente as views deverão estar contidas na pasta “/home”, é apenas uma forma de organizá-las melhor.

**Spring MVC: Views**

Uma das áreas em que o Spring se destaca é a separação das tecnologias de visualização do resto do framework MVC. Por exemplo, decidir usar o Velocity ou o XSLT no lugar de um JSP existente é basicamente uma questão de configuração. Este capítulo aborda as principais tecnologias de visualização que funcionam com o Spring e aborda brevemente como adicionar novos. Este capítulo pressupõe que você já esteja familiarizado com a [Seção 17.5, “Resolvendo visões”,](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-viewresolver) que aborda o básico de como as visões, em geral, são acopladas à estrutura MVC.

O Spring fornece algumas soluções prontas para exibição JSP e JSTL. O uso de JSP ou JSTL é feito usando um resolvedor de visão normal definido no WebApplicationContext. Além disso, é claro que você precisa escrever algumas JSPs que realmente renderizarão a exibição.

[O Velocity](https://velocity.apache.org/) e o [FreeMarker](http://www.freemarker.org/) são duas linguagens de modelagem que podem ser usadas como tecnologias de visualização em aplicativos Spring MVC. Os idiomas são bastante semelhantes e servem a necessidades semelhantes e, portanto, são considerados juntos nesta seção. Para diferenças semânticas e sintáticas entre os dois idiomas, consulte o site do [FreeMarker](http://www.freemarker.org/) .

O XSLT é uma linguagem de transformação para XML e é popular como uma tecnologia de visualização em aplicativos da Web. O XSLT pode ser uma boa escolha como tecnologia de visualização se seu aplicativo lida naturalmente com XML ou se seu modelo pode ser facilmente convertido em XML. A seção a seguir mostra como produzir um documento XML como dados de modelo e transformá-lo com XSLT em um aplicativo Spring Web MVC.

**Spring MVC: REST**

A web e seu protocolo principal, HTTP, fornecem uma pilha de recursos:

* Ações adequadas ( GET, POST, PUT, DELETE, ...)
* Cache
* Redirecionamento e encaminhamento
* Segurança (criptografia e autenticação)

Assim, construindo sobre HTTP, as APIs REST fornecem os meios para construir APIs flexíveis que podem:

* Suportar compatibilidade com versões anteriores
* APIs evolutivas
* Serviços escalonáveis
* Serviços seguráveis
* Um espectro de serviços sem estado a stateful

O que é importante perceber é que o REST, por mais onipresente que seja, não é um padrão,por si só , mas uma abordagem, um estilo, um conjunto de restrições em sua arquitetura que pode ajudá-lo a construir sistemas em escala da web. Neste tutorial, usaremos o portfólio do Spring para criar um serviço RESTful, aproveitando os recursos sem pilha do REST.

**Isso é o que nossa API REST faz:**

* Solicitação **GET** para / api / user / retorna uma lista de usuários
* Solicitação **GET** para / api / user / 1 retorna o usuário com ID 1
* **A** solicitação **POST** para / api / user / com um objeto de usuário como JSON cria um novo usuário
* Solicitação **PUT** para / api / user / 3 com um objeto de usuário como JSON atualiza o usuário com ID 3
* **A** solicitação **DELETE** para / api / user / 4 exclui o usuário com o ID 4
* **EXCLUIR** solicitação para / api / user / exclui todos os usuários

|  |
| --- |
| package com.websystique.springmvc.controller;      @RestController  public class HelloWorldRestController {        @Autowired      UserService userService;  //Service which will do all data retrieval/manipulation work          //-------------------Retrieve All Users--------------------------------------------------------        @RequestMapping(value = "/user/", method = RequestMethod.GET)      public ResponseEntity<List<User>> listAllUsers() {          List<User> users = userService.findAllUsers();          if(users.isEmpty()){              return new ResponseEntity<List<User>>(HttpStatus.NO\_CONTENT);//You many decide to return HttpStatus.NOT\_FOUND          }          return new ResponseEntity<List<User>>(users, HttpStatus.OK);      }          //-------------------Retrieve Single User--------------------------------------------------------        @RequestMapping(value = "/user/{id}", method = RequestMethod.GET, produces = MediaType.APPLICATION\_JSON\_VALUE)      public ResponseEntity<User> getUser(@PathVariable("id") long id) {          System.out.println("Fetching User with id " + id);          User user = userService.findById(id);          if (user == null) {              System.out.println("User with id " + id + " not found");              return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);          }          return new ResponseEntity<User>(user, HttpStatus.OK);      }            //-------------------Create a User--------------------------------------------------------        @RequestMapping(value = "/user/", method = RequestMethod.POST)      public ResponseEntity<Void> createUser(@RequestBody User user,    UriComponentsBuilder ucBuilder) {          System.out.println("Creating User " + user.getName());            if (userService.isUserExist(user)) {              System.out.println("A User with name " + user.getName() + " already exist");              return new ResponseEntity<Void>(HttpStatus.CONFLICT);          }            userService.saveUser(user);            HttpHeaders headers = new HttpHeaders();          headers.setLocation(ucBuilder.path("/user/{id}").buildAndExpand(user.getId()).toUri());          return new ResponseEntity<Void>(headers, HttpStatus.CREATED);      }          //------------------- Update a User --------------------------------------------------------        @RequestMapping(value = "/user/{id}", method = RequestMethod.PUT)      public ResponseEntity<User> updateUser(@PathVariable("id") long id, @RequestBody User user) {          System.out.println("Updating User " + id);            User currentUser = userService.findById(id);            if (currentUser==null) {              System.out.println("User with id " + id + " not found");              return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);          }            currentUser.setName(user.getName());          currentUser.setAge(user.getAge());          currentUser.setSalary(user.getSalary());            userService.updateUser(currentUser);          return new ResponseEntity<User>(currentUser, HttpStatus.OK);      }        //------------------- Delete a User --------------------------------------------------------        @RequestMapping(value = "/user/{id}", method = RequestMethod.DELETE)      public ResponseEntity<User> deleteUser(@PathVariable("id") long id) {          System.out.println("Fetching & Deleting User with id " + id);            User user = userService.findById(id);          if (user == null) {              System.out.println("Unable to delete. User with id " + id + " not found");              return new ResponseEntity<User>(HttpStatus.NOT\_FOUND);          }            userService.deleteUserById(id);          return new ResponseEntity<User>(HttpStatus.NO\_CONTENT);      }          //------------------- Delete All Users --------------------------------------------------------        @RequestMapping(value = "/user/", method = RequestMethod.DELETE)      public ResponseEntity<User> deleteAllUsers() {          System.out.println("Deleting All Users");            userService.deleteAllUsers();          return new ResponseEntity<User>(HttpStatus.NO\_CONTENT);      }    } |

**Explicação detalhada :**

**@RestController** : Primeiro de tudo, estamos usando a nova anotação @RestController do Spring 4. Esta anotação elimina a necessidade de anotar cada método com @ResponseBody. Sob o capô, o @RestController é anotado com @ResponseBody e pode ser considerado como uma combinação de @Controller e @ResponseBody.

**@RequestBody** : Se um parâmetro de método é anotado com @RequestBody, o Spring ligará o corpo da solicitação de HTTP de entrada (para a URL mencionada em @RequestMapping para esse método) a esse parâmetro. Ao fazer isso, o Spring irá [nos bastidores] usar [conversores HTTP Message](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#rest-message-conversion) para converter o corpo da solicitação HTTP em objeto de domínio [deserializar corpo da solicitação para objeto de domínio], com base no cabeçalho ACCEPT ou Content-Type presente na solicitação.

**@ResponseBody** : Se um método é anotado com @ResponseBody, o Spring ligará o valor de retorno ao corpo de resposta HTTP de saída. Enquanto isso, o Spring irá [nos bastidores] usar [conversores HTTP Message](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#rest-message-conversion) para converter o valor de retorno para o corpo da resposta HTTP [serializar o objeto para o corpo da resposta], com base no Content-Type presente no cabeçalho HTTP da solicitação. Como já mencionado, no Spring 4, você pode parar de usar essa anotação.

**ResponseEntity** é um negócio real. Ele representa toda a resposta HTTP. Boa coisa é que você pode controlar qualquer coisa que entra nele. Você pode especificar o código de status, os cabeçalhos e o corpo. Ele vem com vários construtores para transportar as informações que você deseja enviar em resposta HTTP.

**@PathVariable** Esta anotação indica que um parâmetro de método deve ser vinculado a uma variável de modelo URI [aquela em '{}'].

Basicamente, @RestController, @RequestBody, ResponseEntity & @PathVariable são tudo o que você precisa saber para implementar uma API REST no Spring 4. Além disso, o Spring fornece várias classes de suporte para ajudá-lo a implementar algo personalizado.

**MediaType:** Com a anotação @RequestMapping, você pode, adicionalmente, especificar o MediaType a ser produzido ou consumido (usando **produz** ou **consome** atributos) por esse método controlador específico, para restringir ainda mais o mapeamento.

**Spring MVC: Injeção Dependências**

O Spring Framework é um framework muito abrangente. A funcionalidade fundamental fornecida pelo Spring Container é a injeção de dependência. A mola fornece um recipiente leve, por exemplo, o recipiente do núcleo da mola, para injeção de dependência (DI). Este contêiner permite que você injete objetos necessários em outros objetos. Isso resulta em um design no qual a classe Java não é hard-coupled. A injeção na mola é feita através de injeção de incubadora via injeção de construção. Essas classes que são gerenciadas pelo Spring devem estar em conformidade com o padrão JavaBean. No contexto das classes Spring, também são referidos como beans ou beans Spring.

O contentor do núcleo da mola:

* manipula a configuração, geralmente baseada em anotações ou em um arquivo XML (XMLBeanFactory)
* gerencia as classes Java selecionadas através do BeanFactory

O contêiner principal usa a chamada fábrica de beans para criar novos objetos. Novos objetos geralmente são criados como Singletons, se não forem especificados de forma diferente.

A partir do Spring 2.5, é possível configurar a injeção de dependência por meio de anotações. Eu recomendo usar essa maneira de configurar seus beans Spring. O próximo capítulo também descreverá a maneira de configurar isso via XML. Crie um novo projeto Java "de.vogella.spring.di.annotations.first" e inclua os jars spring mínimos necessários em seu classpath. Copie sua classe de modelo do projeto de.vogella.spring.di.model para este projeto. Agora você precisa adicionar anotações ao seu modelo para informar ao Spring quais beans devem ser gerenciados pelo Spring e como eles devem ser conectados. Adicione a anotação @Service ao MySpringBeanWithDependency.java e NiceWriter.java. Defina também com @Autowired no método setWriter que a propriedade "writer" será autoperetada pelo Spring.

|  |  |
| --- | --- |
|  | @Autowired dirá ao Spring para procurar um bean Spring que implemente a interface necessária e coloque-a automaticamente no setter. |

Sob a pasta src crie uma pasta META-INF e crie o seguinte arquivo nesta pasta. Este é o arquivo de configuração do Spring.

|  |  |
| --- | --- |
|  | Você também pode configurar o log4j logger (isso é opcional) copiando o seguinte arquivo para a pasta de origem.  log4j.rootLogger=FATAL, first  log4j.appender.first=org.apache.log4j.ConsoleAppender  log4j.appender.first.layout=org.apache.log4j.PatternLayout  log4j.appender.first.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n |

Após essa configuração, você pode conectar o aplicativo. Crie uma classe principal que leia o arquivo de configuração e inicie o aplicativo.

Se você executar o aplicativo, a classe para o IWriterInterface será inserida na classe de teste. Ao aplicar a injeção de dependência, posso substituir esse gravador por um escritor mais sofisticado. Como resultado, a classe Teste não depende da classe concreta do Writer, é extensível e pode ser facilmente testada.

**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 do manipulador** . 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.

Nas próximas seções, é exatamente isso que vamos analisar - as diferenças entre várias implementações de interceptor.

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.

**Uma nota rápida antes de prosseguirmos - se você quiser pular a teoria e pular direto para os exemplos, pule direto para a seção 5.**

Veja como uma simples implementação do preHandle () será:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Override  public boolean preHandle(    HttpServletRequest request,    HttpServletResponse response,    Object handler) throws Exception {      // your code      return true;  } |

Observe que o método retorna um valor booleano - que informa ao Spring se o pedido deve ser processado por um manipulador ( verdadeiro ) ou não ( falso ).

Em seguida, temos uma implementação de postHandle () :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Override  public void postHandle(    HttpServletRequest request,    HttpServletResponse response,    Object handler,    ModelAndView modelAndView) throws Exception {      // your code  } |

**Esse método é chamado imediatamente após a solicitação ser processada pelo HandlerAdapter , mas antes de gerar uma exibição.**

E é claro que pode ser usado de várias maneiras - por exemplo, podemos adicionar um avatar de um usuário logado em um modelo.

O método final que precisamos implementar na implementação personalizada do HandlerInterceptoré afterCompletion ():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Override  public void afterCompletion(    HttpServletRequest request,    HttpServletResponse response,    Object handler, Exception ex) {      // your code  } |

Quando a visualização é gerada com sucesso, podemos usar esse gancho para fazer coisas como reunir estatísticas adicionais relacionadas à solicitação.

Uma nota final a ser lembrada é que um HandlerInterceptor está registrado no bean DefaultAnnotationHandlerMapping , que é responsável por aplicar os interceptores a qualquer classe marcada com uma anotação @Controller . Além disso, você pode especificar qualquer número de interceptores em seu aplicativo da web.

**Spring MVC: Validadores**

Um validador Spring é uma classe opcional que pode ser chamada para validar dados de formulário para um

dado comando (form) controller.This validador classe é uma classe concreta que implementa

a interface org.springframework.validation.Validator. Um dos dois métodos

requerido por esta interface é o método validate, que é passado a um objeto de comando, como

mencionado anteriormente e um objeto Errors, que pode ser usado para retornar erros. eu vou

demonstrar um exemplo de uma classe Validator mais adiante neste capítulo. Outra validação notável

classe é org.springframework.validation.ValidationUtils, que fornece

métodos convenientes para rejeitar campos vazios.

A partir do Java EE 6 temos uma especificação que resolve este problema. A JSR 303, também conhecida como Bean Validation, define uma série de anotações e uma API para criação de validações para serem utilizadas em Java Beans, que podem ser validados agora em qualquer camada da aplicação.

Com o Bean Validation declaramos através de anotações as regras de validação dentro do nosso modelo, por exemplo, na nossa tarefa:

public class Tarefa {

private Long id;

@Size(min=5)

private String descricao;

private boolean finalizado;

private Calendar dataFinalizacao;

//...

}

Pronto! Com essas anotações, qualquer objeto do tipo Tarefa pode ser validado na camada de controller. Só falta avisar o Spring MVC que realmente queremos executar a validação. Isso é feito pela anotação Valid que devemos usar na antes do parâmetro da ação:

@RequestMapping("adicionaTarefa")

public String adiciona(@Valid Tarefa tarefa) {

JdbcTarefaDao dao = new JdbcTarefaDao();

dao.adiciona(tarefa);

return "tarefa/adicionada";

}

Como estamos falando de Spring MVC, antes da chamada do método é executada a validação, ou seja será verificado se a descrição da tarefa não está vazia. Se estiver, será lançada uma exceção do tipo ConstraintViolationException que possui a descrição do erro.

Não queremos mostrar uma exceção para o usuário e sim apenas voltar para o formulário para mostrar uma mensagem que a validação falhou. O Spring MVC pode guardar o resultado (os erros de validação) em um objeto do tipo BindingResult. Assim não será lançado um exceção. Este objeto BindingResult se torna um parâmetro da ação. Então só é preciso perguntar para ele se existe um erro de validação e se existir, voltar para o formulário. Veja o código:

@RequestMapping("adicionaTarefa")

public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

if(result.hasFieldErrors("descricao")) {

return "tarefa/formulario";

}

JdbcTarefaDao dao = new JdbcTarefaDao();

dao.adiciona(tarefa);

return "tarefa/adicionada";

}

No código acima verificamos se existe um de erro validação relacionado com o atributodescricao da tarefa. Também podemos conferir se existe algum erro de validação, mais genérico:

@RequestMapping("adicionaTarefa")

public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

if(result.hasErrors()) {

return "tarefa/formulario";

}

Para exibir as mensagens de validação no JSP usamos um tag especial que o Spring MVC oferece. O tag se chama **form:errors**:

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

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

Abaixo está o código completo do formulário formulario.jsp da pasta tarefa. Repare que é preciso importar o taglib do Spring MVC:

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

<html>

<body>

<h3>Adicionar tarefas</h3>

<form action="adicionaTarefa" method="post">

Descrição:

<br/>

<textarea rows="5" cols="100" name="descricao"></textarea>

<br/>

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

<br/>

<input type="submit" value="Adicionar"/>

</form>

</body>

</html>

**Spring MVC: Plugins**

Spring Boot

#### CONSTRUA QUALQUER COISA COM INICIALIZAÇÃO DE PRIMAVERA

O Spring Boot é o ponto de partida para a construção de todos os aplicativos baseados no Spring. O Spring Boot foi projetado para colocá-lo em operação o mais rápido possível, com configuração inicial mínima do Spring.

|  |  |
| --- | --- |
| - | Comece em segundos usando o Spring Initializr |
| - | Crie qualquer coisa - API REST, WebSocket, Web, Streaming, Tarefas e muito mais |
| - | Segurança Simplificada |
| - | Suporte rico para SQL e NoSQL |
| - | Suporte de tempo de execução incorporado - Tomcat, Jetty e Undertow |
| - | Ferramentas de produtividade do desenvolvedor, como recarga ao vivo e reinicialização automática |
| - | Dependências curadas que apenas funcionam |
| - | Recursos prontos para produção, como rastreamento, métricas e status de integridade |
| - | Funciona no seu IDE favorito - Spring Tool Suite, IntelliJ IDEA e NetBeans |

Spring Framework

Spring Cloud Data Flow

#### CONECTE QUALQUER COISA

O Spring Cloud Data Flow facilita a criação e orquestração de pipelines de dados nativos da nuvem para casos de uso, como dados ingeridos, análises em tempo real e importação / exportação de dados. O Spring Cloud Data Flow simplifica a conexão de sistemas, fornecendo conectores prontos para uso nos cenários de integração mais comuns.

|  |  |
| --- | --- |
| - | Suporta processamento de dados em fluxos e lote em tempo real |
| - | Ingerir, transformar, analisar e armazenar dados |
| - | Conectores para FTP, RDBMS, Cassandra, RabbitMQ, GemFire, Redis e muito mais |
| - | Suporta middleware moderno de mensagens - Kafka e RabbitMQ |
| - | Desenhador visual de Spring Flo para pipelines |
| - | Painel operacional - métricas, verificações de integridade e gerenciamento remoto |
| - | Plataformas suportadas: Cloud Foundry, Kubernetes, Apache YARN, Apache Mesos |

Spring Security

O Spring Security é uma estrutura poderosa e altamente personalizável de autenticação e controle de acesso. É o padrão de fato para proteger aplicativos baseados em Spring.

O Spring Security é um framework que se concentra em fornecer autenticação e autorização para aplicativos Java. Como todos os projetos do Spring, o poder real do Spring Security é encontrado na facilidade com que ele pode ser estendido para atender aos requisitos personalizados.

## Características

* Suporte abrangente e extensível para autenticação e autorização
* Proteção contra ataques como fixação de sessão, clickjacking, falsificação de solicitação entre sites, etc.
* Integração da API Servlet
* Integração opcional com o Spring Web MVC
* Muito mais…