



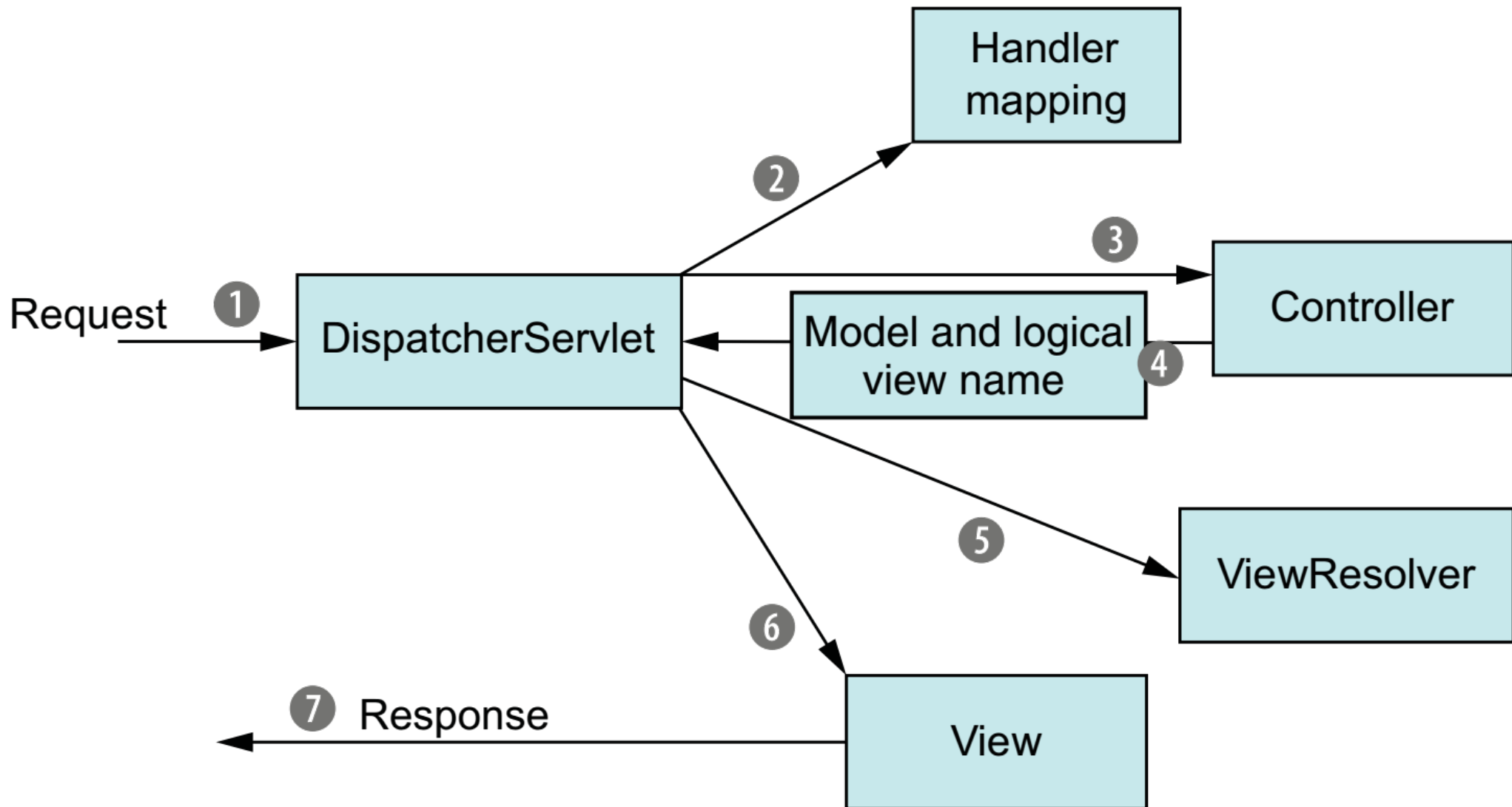
Spring on the Web

Spring na Web

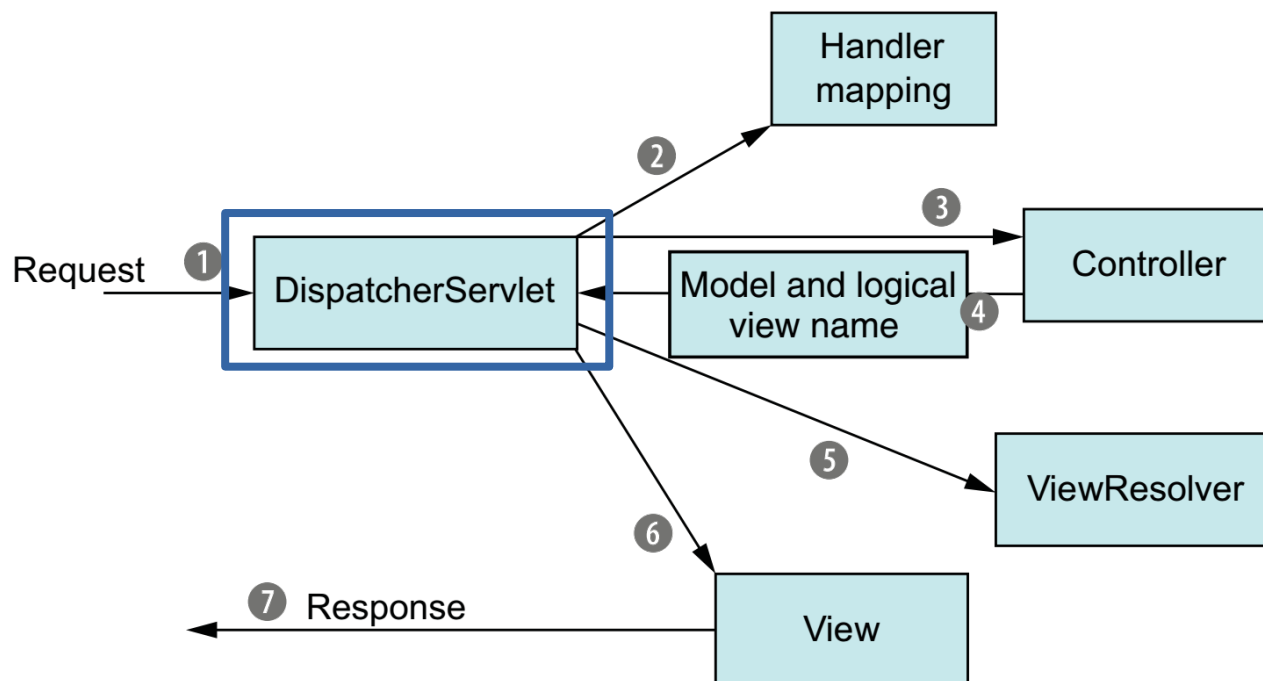
Introdução

- Para muitos desenvolvedores de software com o Java têm como principal objetivo de trabalho o desenvolvimento de *aplicações Web*.
- Nestes tipos de desenvolvimento existem um número significativo de desafios a vencer, como podem ser:
 - A administração do estado, os fluxos de trabalho, e validação.
- O marco de trabalho Web do Spring está desenhado para nos ajudar a alcançar estes elementos.
- O mesmo está apoiado no padrão MVC (Modelo-Vista-controlador)

Ciclo de vida de uma petição HTTP

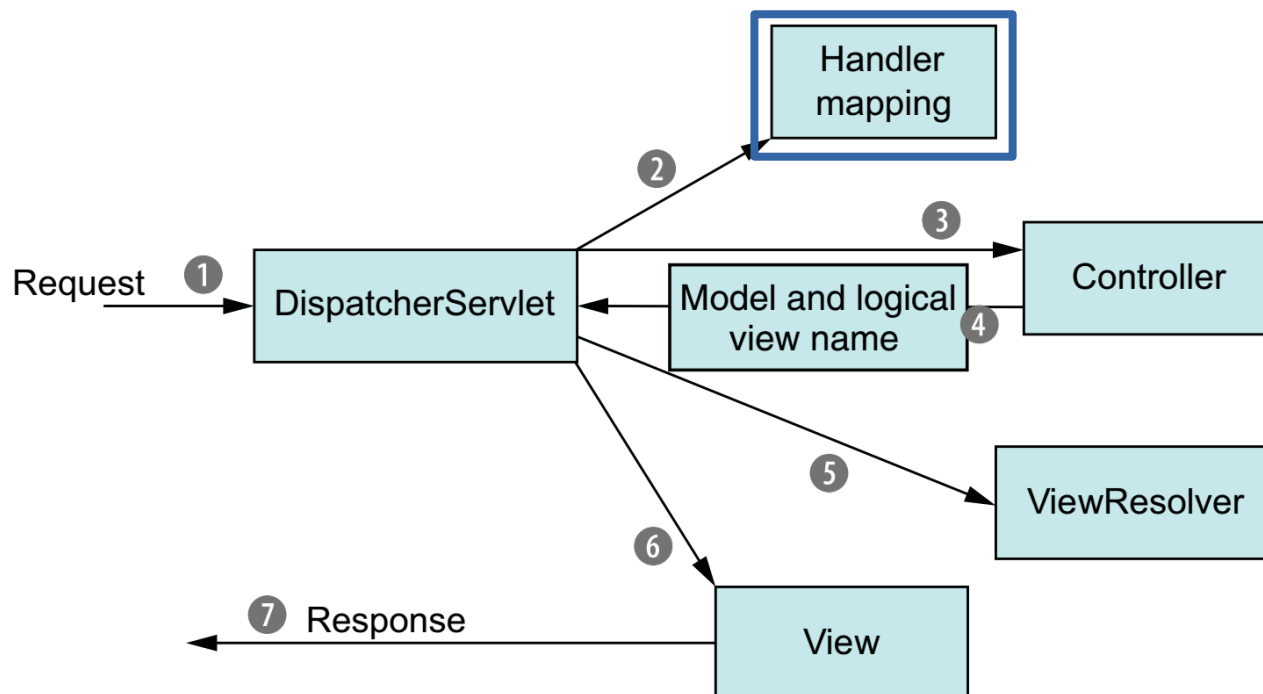


Ciclo de vida de uma petição HTTP



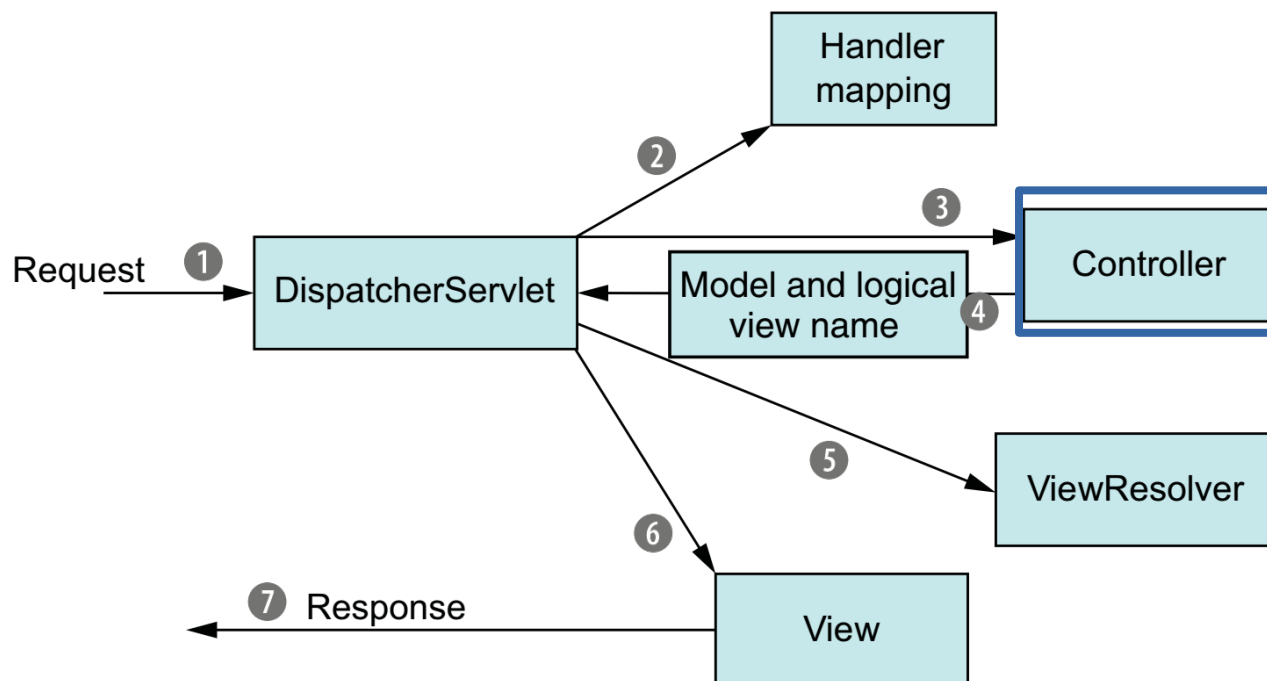
A primeira parada de uma petição é o DispatcherServlet do Spring. Como a maioria de todos os Marcos de trabalhos Web apoiados no MVC todas as petições passam através de só um servlet controlador frontal.

Ciclo de vida de uma petição HTTP



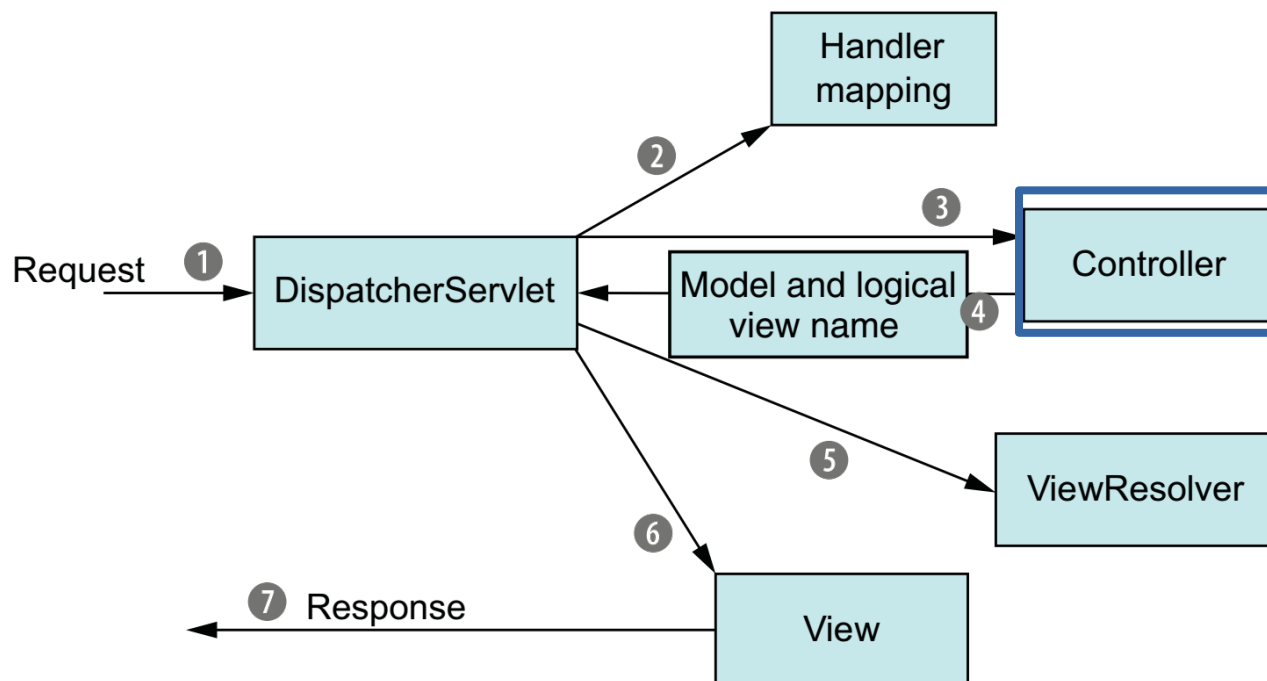
O **DispatcherServlet** consulta um ou mais manipuladores de mapeo para determinar onde será a próxima parada da petição. O manipulador de mapeo disposta uma atenção particular a URL levada pela petição para tomar sua decisão.

Ciclo de vida de uma petição HTTP



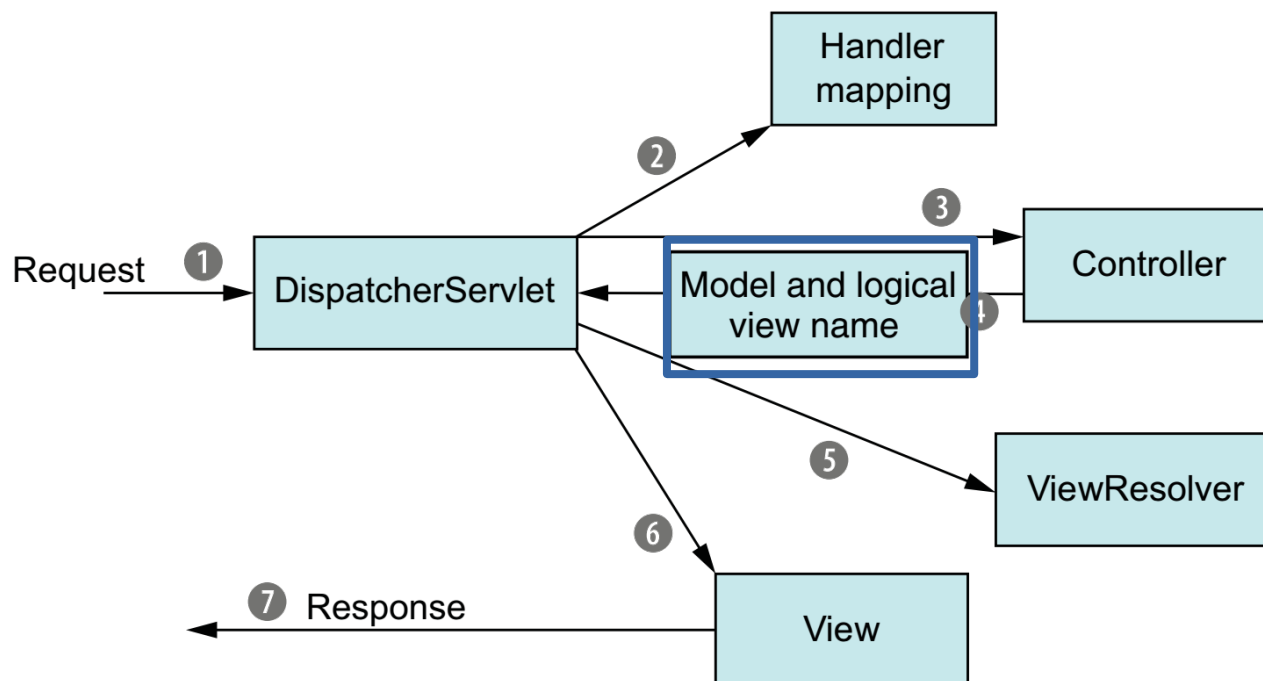
Uma vez que o controlador apropriado foi eleito, o DispatcherServlet envia a petição ao controlador eleito. No controlador, a petição entrega sua carga (a informação enviada pelo usuário) e pacientemente espera enquanto o controlador processa a informação.

Ciclo de vida de uma petição HTTP



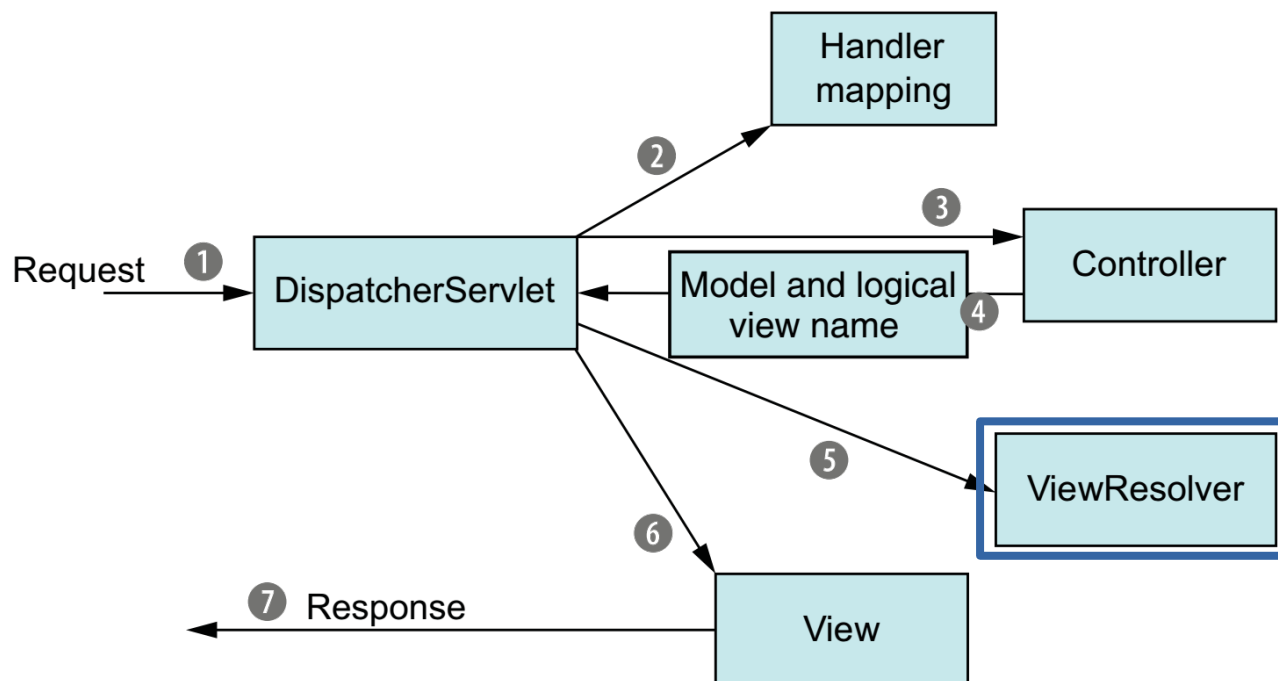
A lógica realizada por um controlador freqüentemente resulta em alguma informação que precisa ser levada de retorno ao usuário e mostrada no navegador. Esta informação é referida como o *modelo*. Mas enviar a informação ao usuário em cru não é suficiente, é necessário que estes dados estejam em um formato amigável para o usuário, tipicamente HTML.

Ciclo de vida de uma petição HTTP



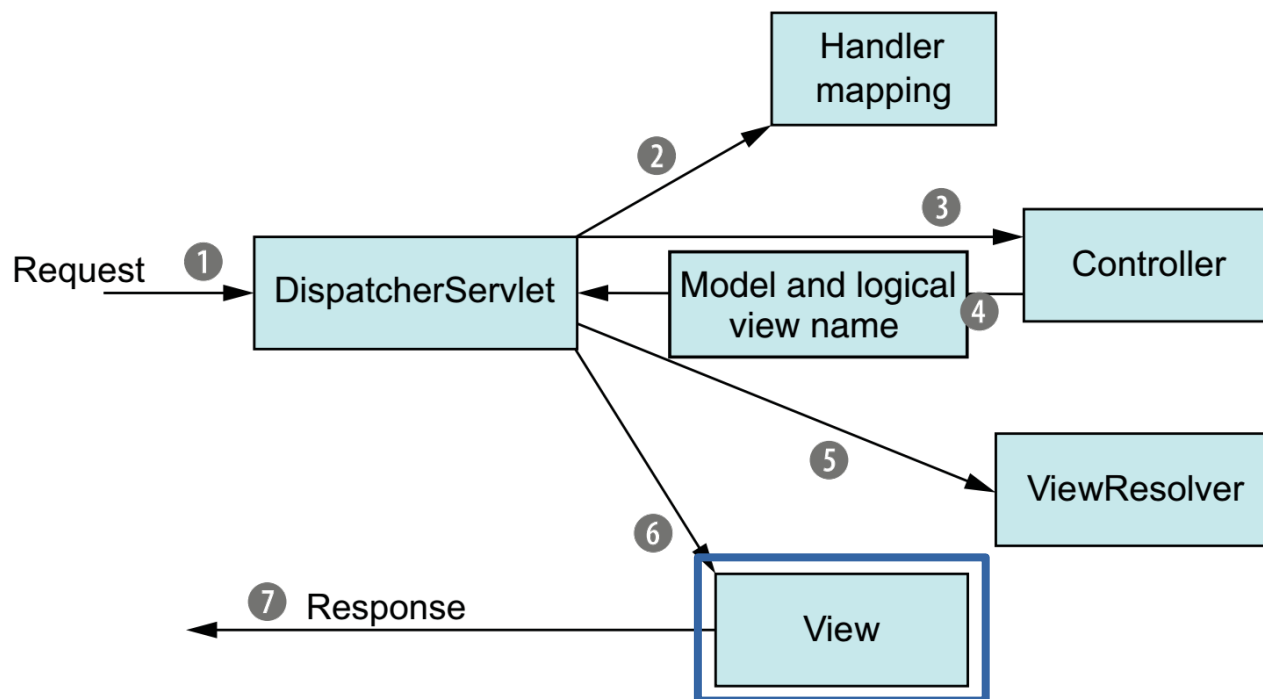
Pelo anterior, precisa-se dar a informação a uma vista, tipicamente uma JavaServer Page (JSP). Uma das últimas coisas que faz um controlador é empacotar os dados do modelo e identificar o nome de uma vista que deve mostrar a saída. E então envia a petição, junto com o modelo e um nome de vista de volta ao DispatcherServlet

Ciclo de vida de uma petição HTTP



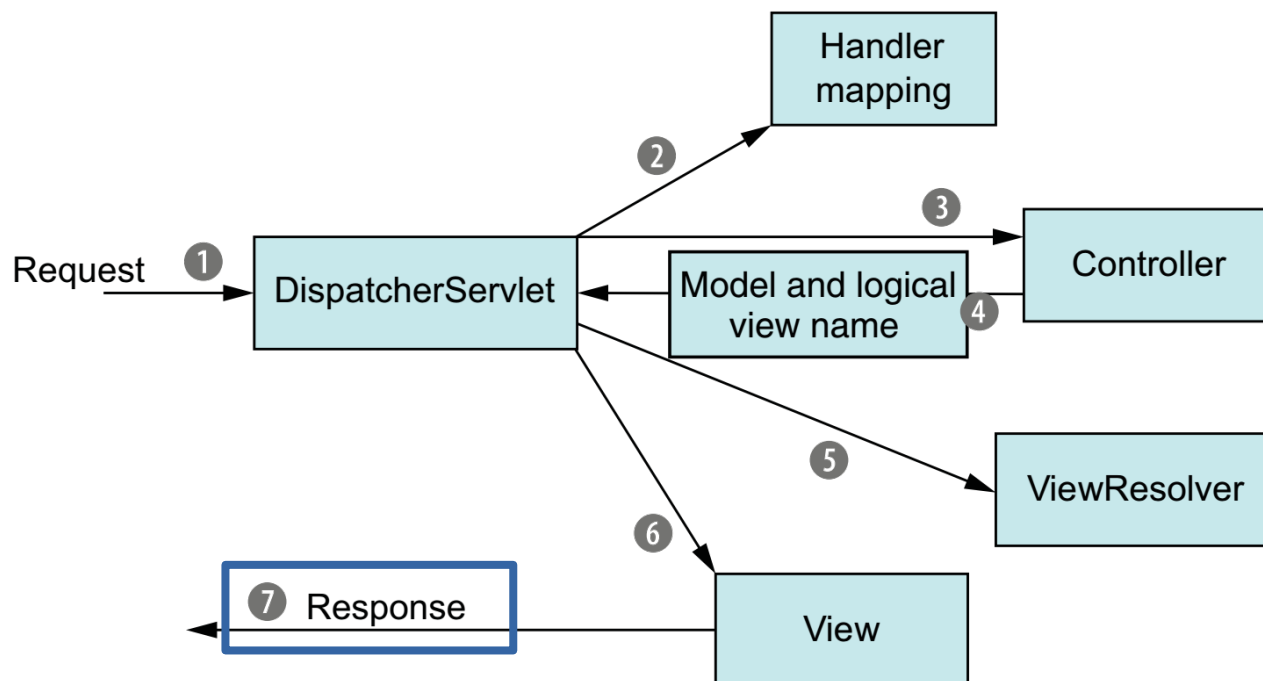
O DispatcherServlet consulta um *resolvedor de vista* para enlaçar um nome de vista lógica com uma implementação específica de vista, a qual pode ou não ser um JSP.

Ciclo de vida de uma petição HTTP



Agora que o DispatcherServlet conhece qual vista vai mostrar o resultado, o trabalho da petição está quase terminado. Sua parada final é na implementação da vista, tipicamente um JSP, onde esta entrega os dados do modelo. Desta maneira o trabalho da petição está finalmente terminado.

Ciclo de vida de uma petição HTTP



A vista usará os dados do modelo para mostrar uma saída que será levada de retorno ao cliente por um objeto de **resposta**.

Configurando o DispatcherServlet

- O DispatcherServlet é a peça central do Spring MVC.
- No é onde a petição chega primeiro no marco de trabalho, e é responsável por enrutar a petição através dos outros componentes.
- Historicamente este componente se configurou no arquivo *web.xml* que é levado nos arquivos WAR das aplicações Web.

Configurando o DispatcherServlet

- Em vez de usar o web.xml, usaremos Java para configurar o DispatcherServlet

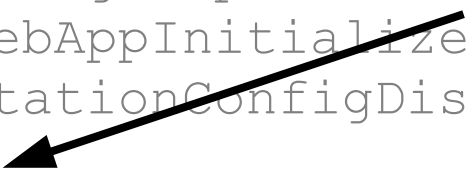
```
package spittr.config;
import org.springframework.web.servlet.support.
AbstractAnnotationConfigDispatcherServletInitializer;
public class SpittrWebAppInitializer
extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { WebConfig.class };
    }
}
```

Configurando o DispatcherServlet

- Em vez de usar o web.xml, usaremos Java para configurar o DispatcherServlet

```
package spittr.config;
import org.springframework.web.servlet.support.
AbstractAnnotationConfigDispatcherServletIniti
public class SpittrWebAppInitializer
extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { WebConfig.class };
    }
}
```

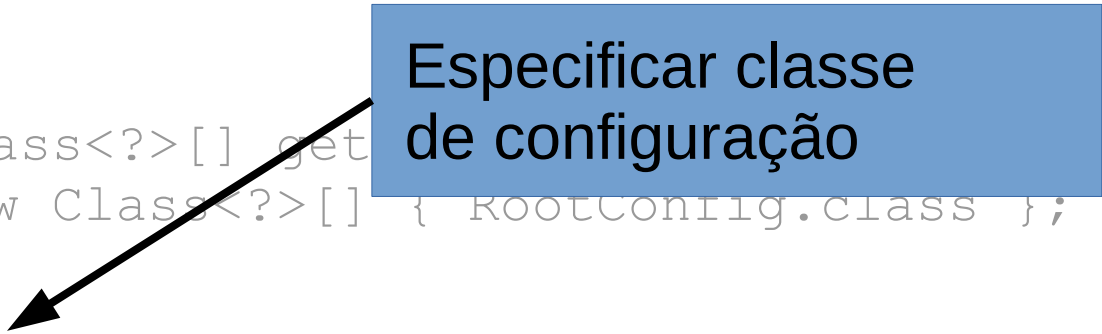
Mapea DispatcherServlet a /



Configurando o DispatcherServlet

- Em vez de usar o web.xml, usaremos Java para configurar o DispatcherServlet

```
package spittr.config;
import org.springframework.web.servlet.support.
AbstractAnnotationConfigDispatcherServletInitializer;
public class SpittrWebAppInitializer
extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
    @Override
    protected Class<?>[] get
        return new Class<?>[] { RootConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { WebConfig.class };
    }
}
```



Especificar classe de configuração

Configurando o DispatcherServlet

- Qualquer classe que estenda do *AbstractAnnotationConfigDispatcherServletInitializer* será utilizado automaticamente para configurar o DispatcherServlet e o contexto de aplicação do Spring.
- Como se pode ver na lâmina anterior a classe *SpittrWebAppInitializer* sobre-escreve três métodos.
 - O primeiro método, `getServletMappings()`, identifica um ou mais caminhos aos que o DispatcherServlet estará engrenado.
 - Neste caso está engrenado a `/`, indicando que este será o servlet por defeito da aplicação. E portanto, processará todas as petições na aplicação.

Configurando o DispatcherServlet

- Nas aplicações Web do Spring há freqüentemente outro contexto de aplicação. Este outro contexto de aplicação é criado pelo `ContextLoaderListener`.
- Enquanto que o *DispatcherServlet* espera que se carreguem os bean que contêm os componentes Web como os controladores, resolvedores de vistas, e manipuladores de mapeos;
- Espera-se que *ContextLoaderListener* carregue os outros bean de sua aplicação.
 - Estes bean são tipicamente os componentes da capa intermédia e a capa de dados que levam o backend da aplicação.

Configurando o DispatcherServlet

- Por debaixo da fachada a classe *AbstractAnnotationConfigDispatcherServletInitializer* cria tanto um *DispatcherServlet* e um *ContextLoaderListener*.
- As classes *@Configuration* retornadas de *getServletConfigClasses()* definirão os beans para o contexto de aplicação do *DispatcherServlet*.
- Enquanto que as classes *@Configuration* que retorna o método *getRootConfigClasses()* será usado para configurar o contexto de aplicação criado pelo *ContextLoaderListener*.

Configurando o DispatcherServlet

- No caso do exemplo a configuração raiz está definida pela classe *RootConfig*, enquanto que a configuração do *DispatcherServlet* está definido na classe *WebConfig*.

Configurando o DispatcherServlet

- **Importante!!!**

- Esta configuração somente é factível quando se desdobra em um servidor que suporta Servlet 3.0; como por exemplo o Apache Tomcat 7 ou maior.
- Se não ser possível então se deve configurar no arquivo web.xml.

Habilitando Spring MVC

- A maneira mais simples de habilitar Spring MVC é criando uma classe cotada com `@EnableWebMvc`.

```
package spittr.config;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.EnableW
ebMvc;
@Configuration
@EnableWebMvc
public class WebConfig {
}
```

Habilitando Spring MVC


- Esta configuração é muito básica e deixa que desejar os seguintes elementos:
 - Não configurou nenhum resolvedor de vistas. E portanto Spring usará `BeanNameViewResolver`. (resolve as vistas procurando um bean cujo ID seja igual no nome da vista e implemente a interface `View`)
 - A busca de componentes não está habilitado. (Spring somente encontrará aqueles controladores que se declarem explicitamente na configuração)
 - Desta maneira o `DispatcherServlet` está mapeado como o servlet por defeito para a aplicação e dirigirá todas as petições, incluindo as petições por recursos estáticos como as imagens e as folhas de estilo.

Habilitando MVC

```
@Configuration
@EnableWebMvc
@ComponentScan("spitter.web")
public class WebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setExposeContextBeansAsAttributes(true);
        return resolver;
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurator) {
        configurator.enable();
    }
}
```

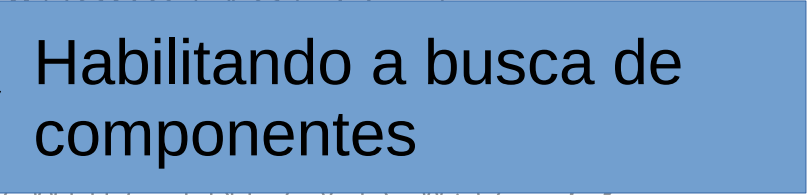
Habilitando MVC

```
@Configuration
@EnableWebMvc
@ComponentScan("spitter.web")
public class WebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setExposeContextBeansAsAttributes(true);
        return resolver;
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```



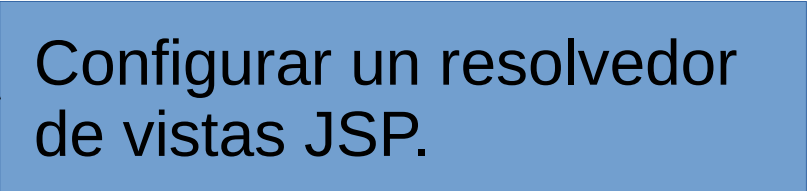
Habilitando MVC

```
@Configuration
@EnableWebMvc
@ComponentScan("spitter.web")
public class WebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setExposeContextBeansAsAttributes(true);
        return resolver;
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```



Habilitando MVC

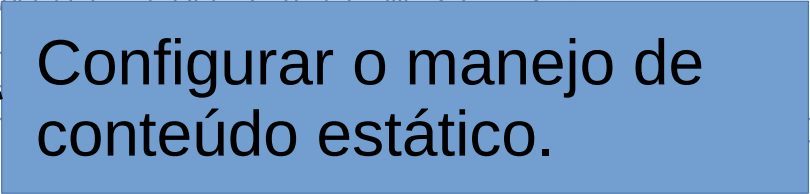
```
@Configuration
@EnableWebMvc
@ComponentScan("spitter.web")
public class WebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setExposeContextBeansAsAttributes(true);
        return resolver;
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```



Configurar un resolvedor de vistas JSP.

Habilitando MVC

```
@Configuration
@EnableWebMvc
@ComponentScan("spitter.web")
public class WebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver resolver =
            new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        resolver.setExposeControllerReferences(true);
        return resolver;
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```



Configurar o manejo de conteúdo estático.

RootConfig

- Devido a que estamos enfocados no desenvolvimento Web, e as configurações Web são realizadas no contexto de aplicação criado pelo DispatcherServlet, manteremos RootConfig relativamente simples por agora.

RootConfig

```
@Configuration
@ComponentScan (
    basePackages={ "spitter" },
    excludeFilters={@Filter (
        type=FilterType.ANNOTATION,
        value=EnableWebMvc.class
    ) }
)
public class RootConfig {
}
```

Escrevendo um controlador simples

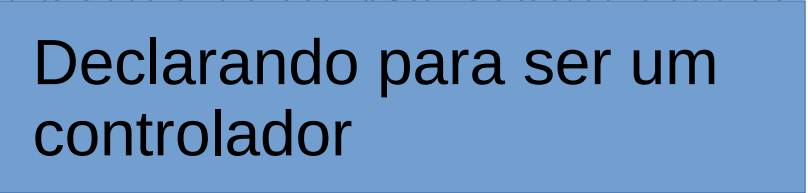
- No Spring MVC, os controladores são só classes com métodos que são cotados com `@RequestMapping` para declarar o tipo de petição que dirigirá.
- Começando com algo simples, imaginemos uma classe controlador que dirija petições para / e mostre a página principal da aplicação.
- *HomeController*, que se mostrará na próxima lâmina, é um exemplo de qual é a maneira mais simples de um controlador do Spring MVC.

Escrevendo um controlador simples

```
package spittr.web;
import static
org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RequestMethod;
@Controller
public class HomeController {
    @RequestMapping(value="/", method=GET)
    public String home() {
        return "home";
    }
}
```

Escrevendo um controlador simples

```
package spittr.web;
import static
org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import
org.springframework.web.servlet.mvc.annotation.annotation.Mapping;
import
org.springframework.web.bind.annotation.RequestMethod;
@Controller
public class HomeController {
    @RequestMapping(value="/", method=GET)
    public String home() {
        return "home";
    }
}
```



Declarando para ser um controlador

Escrevendo um controlador simples

```
package spittr.web;
import static
org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.*;
@Controller
public class HomeController {
    @RequestMapping(value="/", method=GET)
    public String home() {
        return "home";
    }
}
```

Dirige petições de tipo
GET para /

Escrevendo um controlador simples

```
package spittr.web;
import static
org.springframework.web.bind.annotation.RequestMethod.*;
import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RequestMethod;
@Controller
public class HomeController {
    @RequestMapping(value="/", method=GET)
    public String home() {
        return "home";
    }
}
```

O nome da vista é **home**



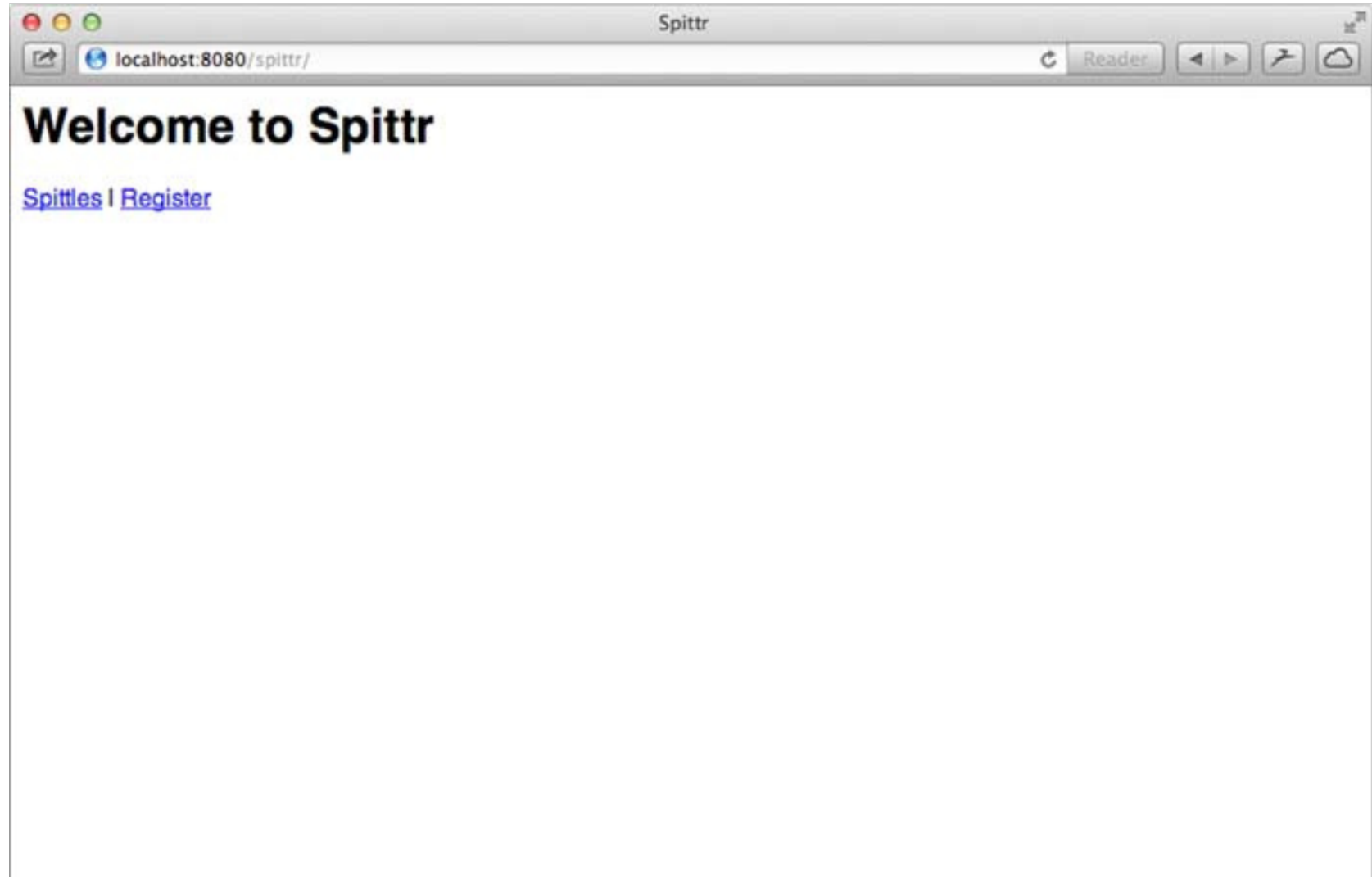
Página principal

- Como se configurou o `InternalResourceViewResolver`, a vista com nome `home` será procurada como um JSP na endereço: `/WEB-INF/views/home.jsp`
- Por agora manteremos sorte página como estática.

Página principal

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
<%@ page session="false" %>
<html>
  <head>
    <title>Spittr</title>
    <link rel="stylesheet" type="text/css"
href="<c:url value="/resources/style.css" />" >
  </head>
  <body>
    <h1>Welcome to Spittr</h1>
    <a href="<c:url value="/spittles"
/>">Spittles</a> |
    <a href="<c:url value="/spitter/register"
/>">Register</a>
  </body>
</html>
```

Página principal



Definindo manejo de petições a nível de classes

- O `@RequestMapping` se pode dividir, colocando a parte da endereço da URL a nível de classe.
- Além disso, pode-se definir uma lista de endereços as colocando em forma de acerto.

Definindo manejo de petições a nível de classes

```
@Controller
@RequestMapping({"/", "/homepage"})
public class HomeController {
    @RequestMapping(method=GET)
    public String home() {
        return "home";
    }
}
```

Definindo manejo de petições a nível de classes

Mapeando o controlador a / e /homepage

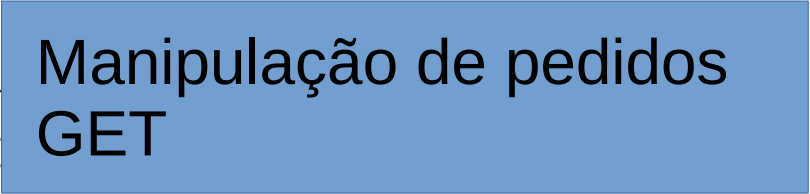


```
@Controller
@RequestMapping("/{", "/homepage"})
public class HomeController {
    @RequestMapping(method=GET)
    public String home() {
        return "home";
    }
}
```


Definindo manejo de petições a nível de classes

```
@Controller
@RequestMapping("/{", "/home")
public class HomeController {
    @RequestMapping(method=GET)
    public String home() {
        return "home";
    }
}
```

Manipulação de pedidos
GET



Definindo manejo de petições a nível de classes

```
@Controller
@RequestMapping("/{", "/homepage"})
public class HomeController
    @RequestMapping(method=
public String home() {
    return "home";
}
}
```

O nome da visão é **home**



Passando dados do modelo à vista

- Primeiro é necessário definir um repositório para acessar aos dados.
- Por fins de desacoplamento, e para não nos preocupar com uma base de dados em específico, definiremos o repositório agora como uma interface e criaremos a implementação da mesma logo.
- Até o momento, somente necessitamos um repositório que devolva uma lista com as entradas do micro blog.

Passando dados do modelo à vista

```
package spittr.data;
import java.util.List;
import spittr.Spittle;
public interface SpittleRepository {
    List<Spittle> findSpittles(long max, int count);
}
```


Spittle
-id: Long -message: String -time: Date -latitude: Double -longitude: Double
+Spittle(message:String,time:Date) +Spittle(message:String,time:Date,longitude:Double latitude:Double) +getId(): Long +getMessage(): String +getTime(): Date +getLongitude(): Double +getLatitude(): Double

Passando dados do modelo à vista

```
@Controller
@RequestMapping("/spittles")
public class SpittleController {
    private SpittleRepository spittleRepository;
    @Autowired
    public SpittleController(
        SpittleRepository spittleRepository) {
        this.spittleRepository = spittleRepository;
    }
    @RequestMapping(method=RequestMethod.GET)
    public String spittles(Model model) {
        model.addAttribute(
            spittleRepository.findSpittles(
                Long.MAX_VALUE, 20));
        return "spittles";
    }
}
```

Passando dados do modelo à vista

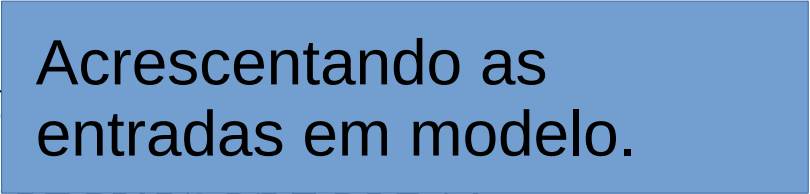
```
@Controller
@RequestMapping("/spittles")
public class SpittleController {
    private SpittleRepository spittleRepository;
    @Autowired
    public SpittleController(
        SpittleRepository spittleRepository) {
        this.spittleRepository = spittleRepository;
    }
    @RequestMapping(method=RequestMethod.GET)
    public String spittles(Model model) {
        model.addAttribute(
            "spittles",
            spittleRepository.findSpittles(
                Long.MAX_VALUE, 20));
        return "spittles";
    }
}
```



Injetando o repositório de dados.

Passando dados do modelo à vista

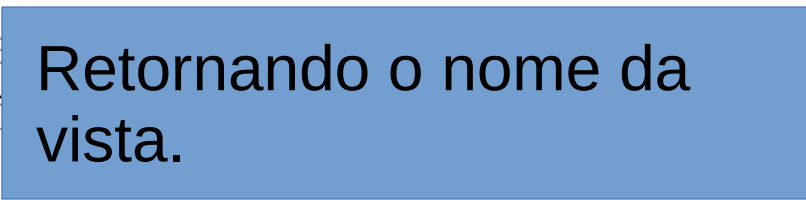
```
@Controller
@RequestMapping("/spittles")
public class SpittleController {
    private SpittleRepository spittleRepository;
    @Autowired
    public SpittleController(
        SpittleRepository spittleRepository) {
        this.spittleRepository = spittleRepository;
    }
    @RequestMapping(method=RequestMethod.GET)
    public String spittles(Model model) {
        model.addAttribute(
            spittleRepository.findSpittles(
                Long.MAX_VALUE, 20)) ;
        return "spittles";
    }
}
```



Acrescentando as entradas em modelo.

Passando dados do modelo à vista

```
@Controller
@RequestMapping("/spittles")
public class SpittleController {
    private SpittleRepository spittleRepository;
    @Autowired
    public SpittleController(
        SpittleRepository spittleRepository) {
        this.spittleRepository = spittleRepository;
    }
    @RequestMapping(method=RequestMethod.GET)
    public String spittles(Model model) {
        model.addAttribute(
            "spittles",
            spittleRepository.findSpittles(
                Long.MAX_VALUE, 20));
        return "spittles";
    }
}
```



Retornando o nome da vista.

Usando o modelo na vista

- A vista deve ser uma página JSP em
/WEB-INF/views/spittles.jsp

```
<c:forEach items="${spittleList}" var="spittle" >
  <li id="spittle_<c:out value="${spittle.id}"/>">
    <div class="spittleMessage">
      <c:out value="${spittle.message}" />
    </div>
    <div>
      <span class="spittleTime">
        <c:out value="${spittle.time}" />
      </span>
      <span class="spittleLocation">
        (<c:out value="${spittle.latitude}" />,
        <c:out value="${spittle.longitude}" />)
      </span>
    </div>
  </li>
</c:forEach>
```

Aceitando petições de entrada

- Spring MVC provê várias maneiras em que um cliente pode acontecer dados aos métodos manipuladores dos controladores. Estes são:
 - Parâmetros de consulta (query parameters)
 - Parâmetros de formulários (form parameters)
 - Variáveis de endereço

Recebendo parâmetros de consulta

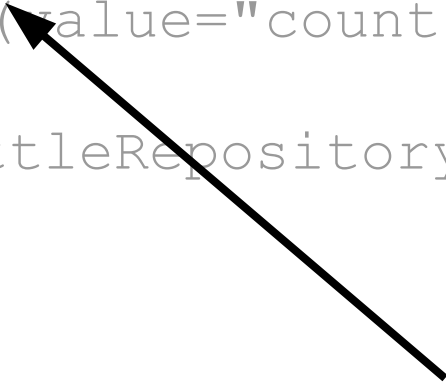
- Uma funcionalidade necessária para a aplicação de exemplo é que o usuário pode ver entradas anteriores às que se mostram na página principal.
- portanto, é necessário passar por parâmetro qual é o conjunto de entradas que se desejam mostrar.
- Para implementar esta funcionalidade é necessário enviar como parâmetros a partir de que entrada se deve mostrar na página e que quantidade.

Recebendo parâmetros

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
    @RequestParam(value="max",
        defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20")
    int count) {
    return spittleRepository.findSpittles(max, count);
}
```

Recebendo parâmetros

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
    @RequestParam(value="max",
        defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20")
    int count) {
    return spittleRepository.findSpittles(max, count);
}
```



Passando parâmetro max,
o valor por defeito. Deve
ser um String.

```
private static final String MAX_LONG_AS_STRING =
    Long.toString(Long.MAX_VALUE);
```

Recebendo parâmetros

```
@RequestMapping(method=RequestMethod.GET)
public List<Spittle> spittles(
    @RequestParam(value="max",
        defaultValue=MAX_LONG_AS_STRING) long max,
    @RequestParam(value="count", defaultValue="20")
    int count) {
    return spittleRepository.findSpittles(max, count);
}
```



Passando o parâmetro
count

Tomando entrada via endereço de entrada

- Digamos que precisamos mostrar uma entrada, dado o ID.
- Uma opção que temos é escrever um manipulador que aceite o ID como um parâmetro de consulta usando `@RequestParam`.

```
@RequestMapping(value="/show", method=RequestMethod.GET)
public String showSpittle(
    @RequestParam("spittle_id") long spittleId,
    Model model) {
    model.addAttribute(
        spittleRepository.findOne(spittleId));
    return "spittle";
}
```

Tomando entrada via endereço de entrada

- Esta maneira poderá dirigir petições com o formato
 - `/spittles/show?spittle_id=12345`
- Embora esta forma funciona, não é a ideal da perspectiva da orientação de recursos.
- Idealmente, o recurso que está sendo identificado (a entrada) deve ser identificado na endereço URL, não por um parâmetro de consulta.
- Como regra geral, os parâmetros de consulta não devem ser usados para identificar um recurso.
- Uma petição GET para </spittles/12345> é melhor que a vista anteriormente.

Tomando entrada via endereço de entrada

- Para acomodar estas variáveis na endereço, Spring MVC permite estabelecer marcadores de posição na endereço @RequestMapping.
- Os marcadores de posição são nomes rodeados por chaves ({ y }).
- As outras partes da endereço precisa coincidir exatamente para a petição a ser dirigida, o marcador pode levar qualquer valor.

Tomando entrada via endereço de entrada

```
@RequestMapping(value="/{spittleId}",  
                  method=RequestMethod.GET)  
public String spittle(  
    @PathVariable("spittleId") long spittleId,  
    Model model) {  
    model.addAttribute(spittleRepository  
                        .findOne(spittleId));  
    return "spittle";  
}
```

Tomando entrada via endereço de entrada

```
<div class="spittleView">  
  <div class="spittleMessage">  
    <c:out value="${spittle.message}" />  
  </div>  
  <div>  
    <span class="spittleTime">  
      <c:out value="${spittle.time}" />  
    </span>  
  </div>  
</div>
```

Processando formulários

- As aplicações Web tipicamente fazem mais que somente lhe mostrar conteúdo ao usuário.
- Principalmente permite aos usuários participar da conversação e encher formulários e enviar dados de volta à aplicação.
- Os controladores do Spring MVC estão tão bem equipados para o processamento de formulários para servir conteúdos.

Processando formulários

- Há dois lados ao processar formulários:
 - Mostrar o formulário
 - Processar os dados que o usuário envia do formulário.
- Na aplicação necessitaremos um formulário para que os usuários novos se registrem.
- Para isso criaremos um controlador *SpitterController*

Processando formulários

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }
}
```

Processando formulários

Dirigir petições GET
para /spitter/register



```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }
}
```

Processando formulários

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
<%@ page session="false" %>
<html>
...
    <form method="POST">
        First Name:
        <input type="text" name="firstName" /><br/>
        Last Name:
        <input type="text" name="lastName" /><br/>
        Username:
        <input type="text" name="username" /><br/>
        Password:
        <input type="password" name="password" /><br/>
        <input type="submit" value="Register" />
    </form>
...
</html>
```


Processando formulários

- Para receber os dados enviados pela petição POST, então é necessário criar um novo método no controlador para que processe os dados enviados.

Processando formulários

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
    private SpitterRepository spitterRepository;
    @Autowired
    public SpitterController(
        SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }
    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }
    @RequestMapping(value="/register", method=POST)
    public String processRegistration(Spitter spitter) {
        spitterRepository.save(spitter);
        return "redirect:/spitter/" + spitter.getUsername();
    }
}
```

Processando formulários


```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
    private SpitterRepository spitterRepository;

    @Autowired
    public SpitterController(
        SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }

    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }

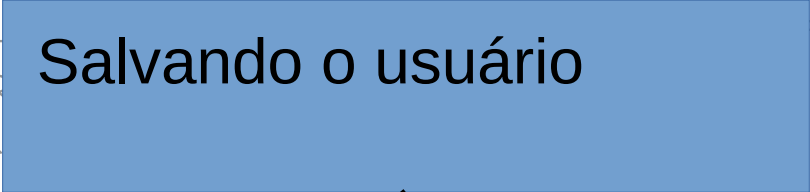
    @RequestMapping(value="/register", method=POST)
    public String processRegistration(Spitter spitter) {
        spitterRepository.save(spitter);
        return "redirect:/spitter/" + spitter.getUsername();
    }
}
```

Injetando a classe SpitterRepository



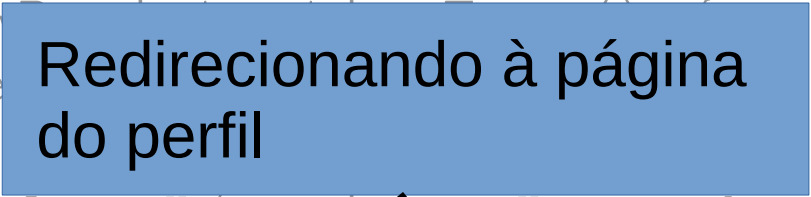
Processando formulários

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
    private SpitterRepository spitterRepository;
    @Autowired
    public SpitterController(
        SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }
    @RequestMapping(value="/show/{username}", method=GET)
    public String showSpitter(@PathVariable String username) {
        return "register";
    }
    @RequestMapping(value="/register", method=POST)
    public String processRegistration(Spitter spitter) {
        spitterRepository.save(spitter);
        return "redirect:/spitter/" + spitter.getUsername();
    }
}
```



Processando formulários

```
@Controller
@RequestMapping("/spitter")
public class SpitterController {
    private SpitterRepository spitterRepository;
    @Autowired
    public SpitterController(
        SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }
    @RequestMapping(value="/register", method=GET)
    public String showRegistrationForm() {
        return "registerForm";
    }
    @RequestMapping(value="/register", method=POST)
    public String processRegistration(Spitter spitter) {
        spitterRepository.save(spitter);
        return "redirect:/spitter/" + spitter.getUsername();
    }
}
```



Processando formulários

- Para processar a petição de redirecionamento devemos criar um método que a dirija.

```
@RequestMapping(value="/{username}", method=GET)
public String showSpitterProfile(
    @PathVariable String username, Model model) {
    Spitter spitter = spitterRepository
        .findByUsername(username);
    model.addAttribute(spitter);
    return "profile";
}
```

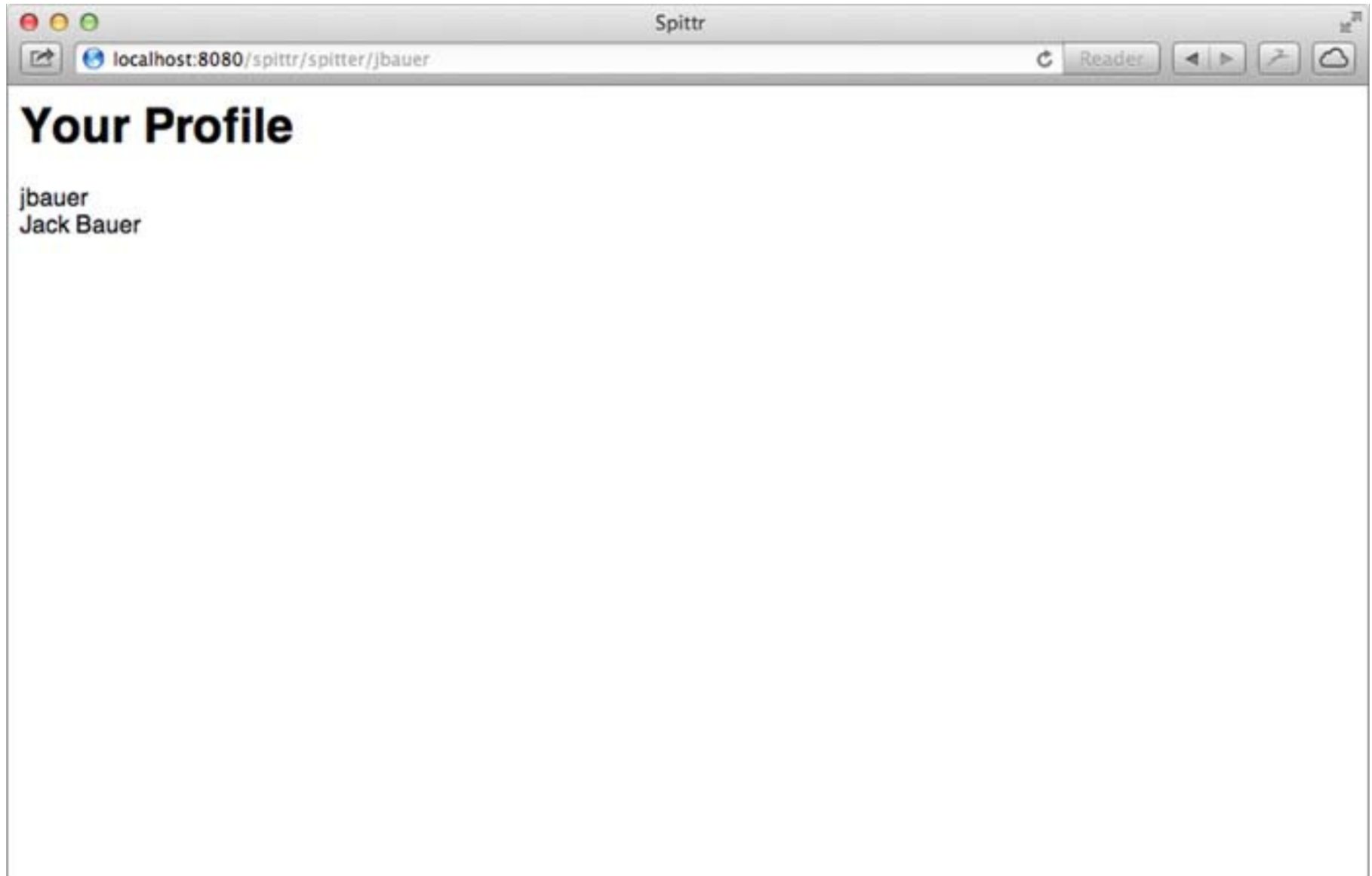
Processando formulários

- Logo devemos criar a vista profile.jsp que mostre a informação do usuário:

```
<h1>Your Profile</h1>  
<c:out value="${spitter.username}" /><br/>  
<c:out value="${spitter.firstName}" />  
<c:out value="${spitter.lastName}" />
```

- Logo se deve mostrar da seguinte maneira:

Processando formulários



Validando formulários

- Que acontece se o formulário envia os campos username e password vazios.
- Isto pode resultar na criação de um novo objeto Spitter no qual ditos campos serão String vazios.
- portanto se deve prevenir que situações estranhas como esta ocorram.
- Spring provê suporte para o [Java Validation API \(JSR-303\)](#). O qual funciona sem ter que realizar configurações extras.

Validando formulários

- O Java Validation API provê uma série de notas que pode pôr nos atributos para acrescentar restrições nos valores desses atributos.
- Todas estas notas estão no pacote `javax.validation.constraints`

Java Validation API

- Entre as anotações que provê dito pacote se encontram:
 - @AssertFalse
 - @AssertTrue
 - @DecimalMax
 - @DecimalMin
 - @Digits
 - @Future
 - @Max
 - @Min
 - @NotNull
 - @Null
 - @Past
 - @Pattern
 - @Size

Validando formulários

- Usando as anotações anteriores se pode modificar o bean do usuário para acrescentar as restrições.

Validando formulários

```
public class Spitter {  
    private Long id;  
    @NotNull  
    @Size(min=5, max=16)  
    private String username;  
    @NotNull  
    @Size(min=5, max=25)  
    private String password;  
    @NotNull  
    @Size(min=2, max=30)  
    private String firstName;  
    @NotNull  
    @Size(min=2, max=30)  
    private String lastName;  
    ...  
}
```

Validando formulários

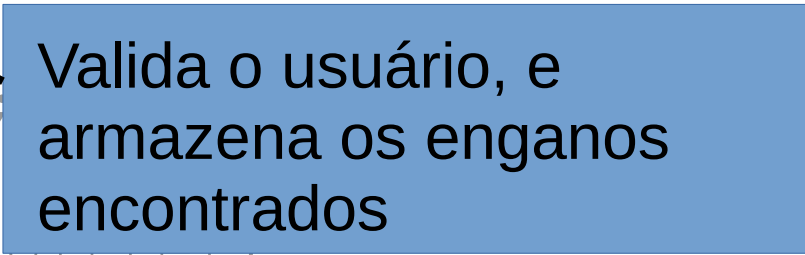
- Agora é necessário modificar o manipulador da petição para que valide.

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @Valid Spitter spitter,
    Errors errors) {
    if (errors.hasErrors()) {
        return "registerForm";
    }
    spitterRepository.save(spitter);
    return "redirect:/spitter/" +
    spitter.getUsername();
}
```

Validando formulários

- Ahora es necesario modificar el manejador de la petición para que valide.

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @Valid Spitter spitter,
    Errors errors) {
    if (errors.hasErrors())
        return "registerForm";
    }
    spitterRepository.save(spitter);
    return "redirect:/spitter/" +
        spitter.getUsername();
}
```

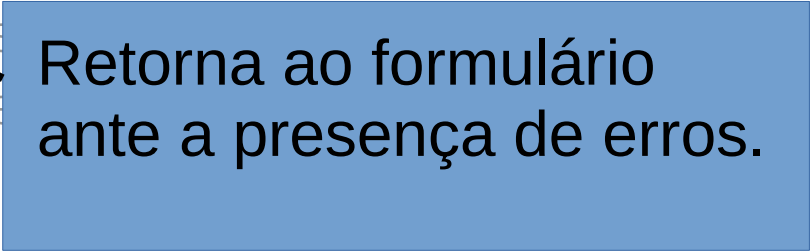


Valida o usuário, e
armazena os enganos
encontrados

Validando formulários

- Ahora es necesario modificar el manejador de la petición para que valide.

```
@RequestMapping(value="/register", method=POST)
public String processRegistration(
    @Valid Spitter spitter,
    Errors errors) {
    if (errors.hasErrors()) {
        return "registerForm";
    }
    spitterRepository.save(spitter);
    return "redirect:/spitters/" +
        spitter.getUsername();
}
```



Retorna ao formulário
ante a presença de erros.

Lecturas recomendadas

- Walls, C. (2014). **Spring in Action** (Fourth Edi). Manning Publications. Retrieved from <https://www.manning.com/books/spring-in-action-fourth-edition>
 - Chapter 6: Rendering web views
 - Chapter 7: Advanced Spring MVC
 - Chapter 9: Securing web applications

Resumem

- Spring possui um poderoso e flexível marco de trabalho Web.
- Empregando anotações, Spring MVC oferece um modelo de desenvolvimento próximo ao POJO. Fazendo simples o trabalho para desenvolver controladores que dirija petições e sejam fácil de provar.
- Viu-se brevemente como escrever vistas para os controladores usando páginas JSP. Mas existem outras tecnologias mais flexíveis.



Spring on the Web

Spring na Web