



VRaptor 3

Felipe Sobreira Cassimiro



Tópicos

- Criação do projeto
- Controlador
- Injeção de dependências
- Escopo
- Controle dos resultados
- Validação
- Integração entre sistemas
- Métodos HTTP e recursos
- Interceptadores
- Conversores



Criação do projeto

Existem várias formas de iniciar um projeto utilizando o VRaptor 3, algumas delas são:

- **VRaptor Blank Project:** já possui a configuração preparada e está configurado para o Eclipse ou NetBeans.
- **Zip do VRaptor:** arquivo com a distribuição completa do framework. Neste arquivo podemos encontrar o .jar do VRaptor e suas dependências, documentação, etc.
- **Maven Project:** após a criação do projeto na IDE, adicionar as dependências do VRaptor no arquivo pom.xml
- **VRaptor Scaffold:** inspirado pela funcionalidade de scaffolding do Ruby on Rails. Para a criação do projeto é necessário ter o RubyGems instalado



Controlador - @Resource

- São as classes que recebem requisições web no sistema
- Uma classe, ou recurso, anotado com @Resource possui todos os métodos acessíveis através de chamadas do tipo GET aURLs específicas
- A classe ao lado, faz com que as URLs seguintes sejam acessíveis e cada uma invocando seu respectivo método:
 - /cliente/adiciona
 - /cliente/lista
 - /cliente/visualiza
 - /cliente/remove
 - /cliente/atualiza

```
@Resource
public class ClienteController {

    public void adiciona(Cliente cliente) {

    }

    public List<Cliente> lista() {
        return ...
    }

    public Cliente visualiza(Cliente perfil) {
        return ...
    }

    public void remove(Cliente cliente) {

    ...
    }

    public void atualiza(Cliente cliente) {
        ...
    }
}
```



Controlador - @Resource

- Quando tivermos formulários, e formos utilizar métodos post para submeter dados para o servidor, podemos aplicar um parâmetro no método da aplicação onde será feito a chamada pelo formulário.
- O objeto será criado a partir destes inputs.
- Como convenção do framework, para que isso ocorra o parâmetro do método deve ter o mesmo prefixo dos inputs do formulário, e as propriedades devem seguir o padrão do modelo criado.
- No caso ao lado, na página do formulário, ao ser chamado o método adicionar ao controlador, o método irá executar a ação implementada e irá redirecionar para a página adicionar.jsp

```
public class Cliente{
    private String nome;
    private int idade;
    private String endereco;
    private ...

    // setter e getters
}

//...
public void adicionar(Cliente cliente) {
    this.lista.add(cliente);
    //...
}

<!-- formulario.jsp -->
<form action="${linkTo[ClienteController].adicionar}" method="post">
    <input type="text" name="cliente.nome" />
    <input type="text" name="cliente.idade" />
    <input type="text" name="cliente.endereco" />
</form>
<!-- etc -->
```



Controlador - @Resource

- Por padrão o retorno dos métodos ficam disponíveis na JSP. Então se temos no retorno do método um objeto do tipo Cliente, esse retorno será colocar numa variável chamada \${cliente}.
- Em uma outra situação, se quisermos retornar uma lista de clientes (List<Cliente>) o nome da variável será \${livroList}.

```
//...  
public List<Cliente> listar() {  
    //...  
    return this.lista;  
}
```

```
<!-- listar.jsp -->  
<h3>Lista de Clientes</h3>  
  
<c:forEach items="${clienteList}" var="cliente">  
    <li>${cliente.nome} - ${cliente.endereco}</li>  
</c:forEach>  
  
<!-- etc -->
```

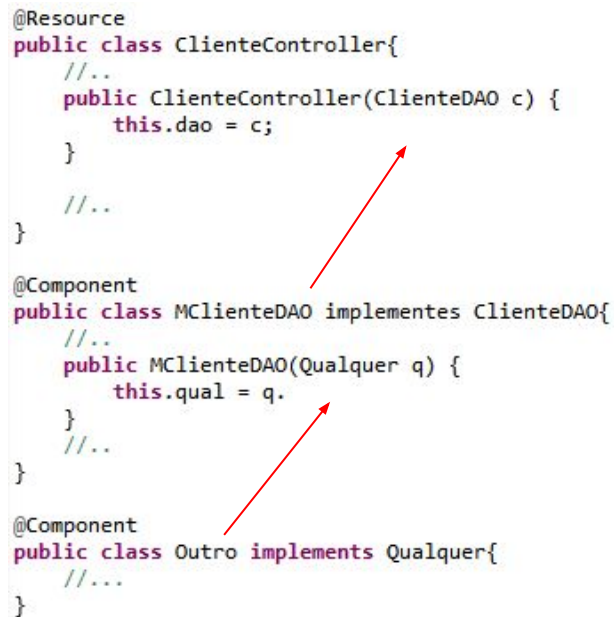
Injeção de dependências

- A dependência é injetada via construtor.
- O VRaptor utiliza esta técnica de inversão de controle, para evitar que uma classe não precise criar e gerenciar suas dependências.
- Para isso, é necessário declarar quais são os componentes necessários para o funcionamento de cada classe. Assim a responsabilidade vai sendo empurrada para camadas inferiores até chegarmos em um componente especializado.
- Cada componente declara suas dependências, e se possível como interfaces para ficarmos livres para usar qualquer implementação disponível.

```
@Resource
public class ClienteController{
    //..
    public ClienteController(ClienteDAO c) {
        this.dao = c;
    }
    //..
}

@Component
public class MClienteDAO implements ClienteDAO{
    //..
    public MClienteDAO(Qualquer q) {
        this.qual = q.
    }
    //..
}

@Component
public class Outro implements Qualquer{
    //...
}
```





Tempo de “vida” dos componentes - Escopo

- Determinamos como os componentes serão criados, mas precisamos saber quando.
- O escopo é o tempo que um componente será usado no sistema.
- Durante um escopo, todas as classes dependentes de um componente, receberão a mesma instância desse componente.
- Toda classe gerenciada pelo VRaptor, como as anotadas com **@Component** ou **@Resource**, é por padrão de escopo de requisição.
- Tipos de escopo:
 - @SessionScoped: escopo de sessão, objeto atrelado à sessão HTTP do usuário
 - @ApplicationScoped: escopo de aplicação, apenas uma instância para a aplicação
 - @PrototypeScoped: escopo de protótipo, uma instância diferente para cada dependência
 - @RequestScoped: escopo de requisição, cada request terá uma instância diferente
- Obs.: Um componente não pode ser dependência de um outro de escopo maior.
- Um método anotado como **@PreDestroy** será executado antes do objeto ser destruído.



Controle dos resultados

- A Classe Result, é o componente do VRaptor responsável pela personalização do resultado final da execução de um método do controller.
- Podemos injetá-lo pelo método, ou pelo construtor da classe caso queiramos utilizar em mais de um método.
- Com este novo artefato, podemos adicionar mais de uma variável de retorno em um método.

```
@Resource
public class ClienteController{
    //..
    public ClienteController(ClienteDAO c, Result s) {
        this.dao = c;
        this.result = s;
    }

    //..
}

//..
public List<Cliente> listar(){
    //..
    this.result.include("clientes", this.dao);
    this.result.include("mensagem", "etc [..]");
    this.result.include("outro", this.outro);
}
```

Controle dos resultados

- Com o componente Result, também podemos fazer redirecionamento de um método para outro.
- As formas de redirecionamento são:
 - `result.of(this).metodo()`: a página do método indicada será renderizada sem executar o método, indicado quando queremos compartilhar a mesma jsp.
 - `result.forwardTo(this).metodo()`: executado do lado do servidor, a URL é transparente para o cliente.
 - `result.redirectTo(this).metodo()`:
- Obs.: o parâmetro dos método de redirecionamento pode ser a classe de outro controller. ex.: `HomeController.class`

```
@Path("/clientes/exclui/{id}")
public void exclui(String id) {
    this.estante.exclui(id);

    result.include("mensagem", "Cliente excluido com sucesso!");
    result.forwardTo(this).lista();
}
```

Estamos em:

- `http://localhost:8080/livraria-admin/clientes`

passamos para:

- `http://localhost:8080/livraria-admin/clientes/exclui/23`

e continuamos na página da lista.

Ao atualizar a página, os métodos `lista()` e `exclui()` serão executados novamente.

```
@Path("/clientes/exclui/{id}")
public void exclui(String id) {
    // [...] utilizando o redirectTo
    result.redirectTo(this).lista();
}
```

Estamos em:

- `http://localhost:8080/livraria-admin/clientes`

passamos para:

- `http://localhost:8080/livraria-admin/clientes`

Percebemos que o

Ao atualizar a página, somente o método `lista()`

é executado novamente



Controle dos resultados

- O componente Result possui um método chamado use() que nos permite usar diversos tipos de resultados, da seguinte forma:
 - result.use(umTipoDeResultado).configuracaoDesseResultado();
- Os tipos de resultados que já vêm implementado no VRaptor estão disponíveis através de métodos estáticos na classe Results. Alguns dos resultados são:
 - Results.http()
 - Results.status()
 - Results.json()
 - etc.

```
result.use(Results.json()).from(cliente).serialize();
```

```
{ "cliente": {  
  "nome": "Fulano de tal",  
  "endereco": "Avenida XII",  
  "idade": "22"  
} }
```



Validação

- De uma forma clássica podemos fazer validação utilizando condicionais e utilizar o componente **Validator** para adicionar mensagens de erro e redirecionar requisições.
- O validador pode ser adicionado através do construtor da classe.
- Na página de redirecionamento do erro, todos os erros estarão disponíveis na variável `${errors}`.

```
public void adicionar(Cliente cliente) {
    if(cliente.getNome() == null) {
        this.validator.add(
            new ValidationMessage("nome obrigatorio", "nome"));
    }

    if(cliente.getIdade() == null) {
        //...
    }

    this.validator.onErrorRedirectTo(this).etc();

    //Aqui deveria adicionar ....etc
}

<!-- formulario.jsp -->

<c:forEach items="${errors}" var="error">
    <li>${error.category}: ${error.message}</li>
</c:forEach>

<!-- etc -->
```



Validação

- Podemos utilizar a internacionalização de mensagens, e para isso devemos ter um conjunto de arquivos .properties, um para cada língua.
- Para a utilização das mensagens nesses arquivos, precisamos instanciar a classe I18nMessage, como mostra ao lado.

```
messages.properties    => língua padrão  
messages_en.properties => inglês  
messages_es.properties => espanhol
```

```
public void adicionar(Cliente cliente) {  
    if(cliente.getNome() == null) {  
        this.validator  
            .add(new I18nMessage("nome", "campo.obrigatorio"));  
    }  
    //...  
}  
  
//..  
new I18nMessage("idade", "campo.maior.que", "Idade", 0)
```

message.properties:

```
1 campo.obrigatorio = Deve ser preenchido!  
2 campo.maior.que = {0} deve ser maior que {1}  
3 is_not_a_valida_number = "{0}" não é um número válido!  
4 login.ou.senha.invalidos = Login ou senha inválidos !  
5 dinheiro_invalido = "{0}" não é um dinheiro válido !
```



Validação

- Podemos melhorar a nossa validação, anotando os atributos do modelo que deve ser validado. O java possui uma especificação chamada Beans Validation para fazer esse trabalho.
- Nos casos em que forem utilizados mensagens por chave na validação do modelo, é necessário que as mensagens estejam no arquivo **ValidationMessages.properties**

```
import javax.validation.constraints.DecimalMin;
import javax.validation.constraints.NotNull;

public class Cliente{

    @NotNull(message = "Deve preencher !")
    private String nome;

    @NotNull(message = "{campo.obrigatorio}")
    private String endereco;

    @DecimalMin("0.0")
    private float credits;

    //setter e getters...
}
```



Integração entre sistemas

- Precisamos ter uma maneira de fornecer dados para outros sistemas.
- Não podemos enviar objetos através de requisições entre sistemas diferentes. Para resolver essa situação teremos que serializar o objeto a ser enviado.
- Com isso, precisamos de um controller, um objeto serializado, e uma conexão HTTP.

```
@Resource
public class IntegracaoController{
    private ClienteDAO dao;
    private Result result;

    public IntegracaoController(
        ClienteDAO dao
        Result result) {

        this.dao = dao;
        this.result = result;
    }

    public void forneceClientes() {
        List<Cliente> lista = dao.getClientes();

        result.use(Results.xml())
            .from(lista, "clientes").serialize();
    }
}
```



Integração entre sistemas

- Após preparado o caminho para fornecimento dos dados para integração com outros sistemas, é necessário que o sistema cliente também tenha as implementações necessários para ocorrer a comunicação.
- Precisamos implementar o modelo, e a persistência (na memória) do lado do cliente.

```
@Resource
public class HomeController{
    private Acervo dao;
    private Result result;

    public IntegracaoController(
        Acervo dao
        Result result) {

        this.dao = dao;
        this.result = result;
    }

    public void listar() {
        this.result.include(
            "clientes",
            acervo.getClientes());
    }
}
```




Integração entre sistemas

- Precisamos de algo que consiga consumir um serviço HTTP, na URL que nos retorna o XML dos livros.
- A interface ClienteHTTP possui uma implementação que se comunica com o serviço, e transforma os dados capturados em uma string.
- Com a string que representa o objeto serializado em xml, precisamos deserializar para poder utilizar no cliente. Para isso iremos utilizar o XStream.
- Para conseguir consumir o xml, o XStream precisa entender a configuração, então passamos o nó raiz que representa a lista ("clientes") e o nó que representa os elementos ("cliente")

```
@Component
public class AcervoNoAdmin implements Acervo {

    private ClienteHTTP http;

    public AcervoNoAdmin(ClienteHTTP http) {
        this.http = http;
    }

    @Override
    public List<Cliente> getClientes() {

        String url =
            "http://localhost:8080/livraria-admin/integracao/forneceClientes";

        String resposta = http.get(url);

        XStream xstream = new XStream();
        xstream.alias("clientes", List.class);
        xstream.alias("cliente", Livro.class);

        List<Cliente> lista = (List<Cliente>) xstream.fromXML(resposta);

        return livros;
    }
}
```



Métodos HTTP e recursos

- Voltando ao controller do sistema administrador, iremos customizar o mapeamento de todos seus métodos com caminhos e verbos HTTP.
- Por padrão os navegadores aceitam métodos GET e POST, caso queira utilizar outro recurso HTTP, é necessário declarar algum meio que envie a solicitação de outro método para o servidor.
- Podemos utilizar o mesmo caminho para um método do controlador, desde que tenham verbos HTTP diferentes, ou estejam declaradas as prioridades.

```
@Resource
public class ClienteController{

    @Get
    @Path("/clientes/formulario")
    public void formulario() {}

    @Get
    @Path("/clientes")
    public void lista() {
        //..
        this.result.include
            ("livros", this.dao.getClientes());
    }

    @Post
    @Path("/clientes")
    public void adiciona() {
        // ..persistência
        result.redirectTo(this).lista();
    }

    @Get
    @Path(value="/clientes/{stringID}", priority=Path.LOWEST)
    public void edita() {
        //..busca cliente
        result.include(clienteEncontrado);
        result.of(this).formulario();
    }

    @Delete
    @Path("/clientes/{stringID}")
    public void exclui() {}
}
```



Interceptadores

- Funcionam como filtros, possibilitando executar lógicas antes e depois das requisições.
- Para a criação de um interceptor, é necessário implementar a interface de mesmo nome.
- A interface possui dois métodos, o **accepts()** que indica se o objeto(método) interceptado deve ou não ser executado, e o **intercept()** que recebe a pilha de interceptores, para poder continuar a ação do método após ter executado a interceptação

```
@Intercepts
public class ExInterceptor implements Interceptor{

    //..

    @Override
    public boolean accepts(ResourceMethod method) {
        return true;
    }

    @Override
    public void intercept(
        InterceptorStack stack,
        ResourceMethod method,
        Object controller)
        throws InterceptionException {

        //.. Execução antes

        stack.next(method, controller);

        //.. Execução depois

    }
}
```



Interceptadores

- Podemos manipular o que será interceptado, e uma maneira interessante seria criar anotações para indicar o método alvo.
- A classe da anotação deve usar a palavra chave `@interface`.
- É necessário indicar em que lugares essa anotação será válida utilizando a anotação `@target`.
- Também é necessário declarar quando nossa anotação será lida, e para essa situação será em tempo de compilação.
- Podemos utilizar a anotação `@Lazy`, para instanciarmos o interceptador somente quando necessário. (instância funcional somente para executar o `accepts`)

```
@Retention(RetentionPolicy.RUNTIME)
@Target(value = { ElementType.METHOD })
public @interface Integracao{

}

//..no interceptador

@Override
public boolean accepts(ResourceMethod method) {
    return method.containsAnnotation(Integracao.class);
}
```



Conversores

- O VRaptor possui um conjunto de conversores implementados como: String, números, boolean, enum, e datas.
- Podemos criar um conversor personalizado para tratarmos casos específicos.
- Para a criação do conversor, devemos criar uma classe anotada com `@Convert(exemplo.class)` que implementa `Converter<exemplo>`
- A string do método `convert` é o que nos interessa, é o valor captado da requisição para ser convertido.

```
@Convert(Dinheiro.class)
public class DinheiroConverter implements Converter<Dinheiro> {

    @Override
    public Dinheiro convert(
        String value,
        Class<? extends Dinheiro> type,
        ResourceBundle bundle) {

        if (Strings.isNullOrEmpty(value)) { return null;}

        //..ex.: value = R$ 30,00
        for (Moeda moeda : Moeda.values()) {
            if (value.startsWith(moeda.getSimbolo())) {
                return new Dinheiro(moeda, criaMontante(value, moeda, bundle));
            }
        }

        //..throw exceção ConversionError
    }

    private BigDecimal criaMontante(
        String value,
        Moeda moeda,
        ResourceBundle bundle) {

        //..trata o input value
    }
}
```



Referências

- CAVALCANTI, L. VRaptor: Desenvolvimento ágil para web com java
- VRaptor 3, Documentação. Disponível em: <http://vraptor3.vraptor.org/pt/docs>