VRaptor 3 – Java web MVC framework for fast and maintenable development

http://vraptor.caelum.com.br

Automatically generated by Caelum Objects Tubaina
22 de outubro de 2009

Índice

Version: 11.7.22

VRaptor3 - One minute guide

VRaptor 3 focuses in simplicity and, therefore, all of its functionalities have as main goal solve the developer's problem in the less intrusive way.

Either CRUD operations or more complex functionalities such as download, upload, results in different formats (xml, json, xhtml etc), everything is done through VRaptor3's simple and easy-to-understand functionalities. You don't have to deal directly with `HttpServletRequest`, `Responses` or any `javax.servlet` API, although you still have the control of all Web operations.

1.1- Starting up

You can start your project based on vraptor-blank-project, available on http://vraptor.caelum.com.br/download.jsp. It contains all required jar dependencies, and the minimal web.xml configuration for working with VRaptor.

1.2- A simple controller

Having VRaptor properly configured on your web.xml, you can create your controllers for dealing with web requests and start building your system.

A simple controller would be:

```
/*
 * You should annotate your controller with @Resource, so all of its public methods will
 * be ready to deal with web requests.
 */
@Resource
public class ClientsController {

    private ClientDao dao;

    /*
     * You can get your class dependencies through constructor, and VRaptor will be in charge
     * of creating or locating these dependencies and manage them to create your controller.
     * If you want that VRaptor3 manages creation of ClientDao, you should annotate it with
     * @Component
     */
    public ClientsController(ClientDao dao) {
        this.dao = dao;
    }

    /*
     * All public methods from your controller will be reachable through web.
     * For example, form method can be accessed by URI /clients/form,
     * and will render the view /WEB-INF/jsp/clients/form.jsp
     */
```

```
    public void form() {
        // code that loads data for checkboxes, selects, etc
    }

    /*
     * You can receive parameters on your method, and VRaptor will set your parameters
     * fields with request parameters. If the request have:
     * custom.name=Lucas
     * custom.address=Vergueiro Street
     * VRaptor will set the fields name and address of Client custom with values
     * "Lucas" and "Vergueiro Street", using the fields setters.
     * URI: /clients/add
     * view: /WEB-INF/jsp/clients/add.jsp
     */
    public void add(Client custom) {
        dao.save(custom);
    }

    /*
     * VRaptor will export your method return value to the view. In this case,
     * since your method return type is List<Clients>, then you can access the
     * returned value on your jsp with the variable ${clientList}
     * URI: /clients/list
     * view: /WEB-INF/jsp/clients/list.jsp
     */
    public List<Client> list() {
        return dao.listAll():
    }

    /*
     * If the return type is a simple type, the name of exported variable will be
     * the class name with the first letter in lower case. Since this method return
     * type is Client, the variable will be ${client}.
     * A request parameter would be something like id=5, and then VRaptor is able
     * to get this value, convert it to Long, and pass it as parameter to your method.
     * URI: /clients/view
     * view: /WEB-INF/jsp/clients/view.jsp
     */
    public Client view(Long id) {
        return dao.load(id);
    }
}
```

Note this class is independent of `javax.servlet` API. The code is also very simple and can be unit tested easily. VRaptor will make associations with these URIs by default:

```
/client/form   invokes form()
/client/add    invokes add(client) populating the client with request parameters
/clients/list  invokes list() and returns ${clientList} to JSP
/clients/view?id=3  invokes view(3l) and returns ${client} to JSP
```

We'll see later how easy it is to change the URI `/clients/view?id=3` to the more elegant `/clients/view/3`.

ClientDao will also be injected by VRaptor, as we'll see. You can see now the Ten minutes guide.

VRaptor3 - Ten minutes guide

2.1- Starting a project: a on-line store

Let's start by downloading the *vraptor-blank-project* from http://vraptor.caelum.com.br/download.jsp. This blank-project has the required configuration on `web.xml` and the dependencies on `WEB-INF/lib` that are needed to start using VRaptor. You can also import this project on Eclipse.

We need to change the base package using this config at `web.xml`:

```
<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <param-value>com.companyname.projectname</param-value>
</context-param>
```

On the example, all classes from your application **must** be in a subpackage of `com.companyname.projectname`, so VRaptor can find your components and manage your dependencies. In this example project the base package will be:

```
<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <param-value>br.com.caelum.onlinestore</param-value>
</context-param>
```

Assuming that the context root of the application was changed to `/onlinestore`, if you run this example you should be able to access http://localhost:8080/onlinestore and see an **It works!** on screen.

2.2- Product registry

Let's start the system with a products registry. We need a class that will represent the products, and we'll use it to persist products on the database, with Hibernate:

```
@Entity
public class Product {
    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String description;
    private Double price;
    //getter and setters
}
```

We also need a class that will *control* the products' register, handling web requests. This class will be the Products `Controller`:

```
public class ProductsController {
}
```

ProductsController will expose URIs to be accessed through web, i.e, will expose resources of your application. And for indicate it, you must annotate it with `@Resource`:

```
@Resource
public class ProductsController {
}
```

By using this annotation, all public methods of the annotated class will be reachable through web. For instance, if there is a `list` method on the class:

```
@Resource
public class ProductsController {
    public List<Product> list() {
        return new ArrayList<Product>();
    }
}
```

Then, VRaptor will automatically redirect all requests to the URI `/products/list` to this method. The convention for URIs is: `/<controller_name>/<method_name>`.

At the end of method execution, VRaptor will `dispatch` the request to the jsp at `/WEB-INF/jsp/products/list.jsp`. The convention for the default view is `/WEB-INF/jsp/<controller_name>/<method_name>.jsp`.

The `list` method will return a product list, so how can I get it on jsp? On VRaptor, the method return value will be exported to the jsp by request attributes. In this case, the name of the exported attribute will be `productList`, holding the method returned value:

list.jsp

```
<ul>
<c:forEach items="${productList}" var="product">
    <li> ${product.name} - ${product.description} </li>
</c:forEach>
</ul>
```

The convention for the attribute names is pretty intuitive: if it is a collection, as it is the case, the name will be `<collection_type>List`; if it is any other type, the name will be the class name with the first letter in lowercase, i.e, if the type is `Product`, the name will be `product`.

2.3- Creating ProductDao: Dependency Injection

VRaptor widely uses the Dependency Injection and Inversion of Control concept. The whole idea is simple: if you need a resource, you won't create it, but will have it ready for you when you ask for it. You can get more information about it on the Dependency Injection chapter.

We are returning a hard coded empty list on our list method. It would be more helpful if we return a real list, for example all of registered products of the system. In order to do that, let's create a product DAO, for listing the products:

```
public class ProductDao {

    public List<Product> listAll() {
        return new ArrayList<Product>();
    }

}
```

And in the ProductsController we might use the dao for listing products:

```
@Resource
public class ProductsController {

    private ProductDao dao;

    public List<Product> list() {
        return dao.listAll();
    }

}
```

We could create a new ProductDao inside the controller, but we can simply loose coupling by receiving it on the class constructor, and letting VRaptor do its Dependency Management Magic and provide an instance of ProductDao when creating our controller! And for enabling this behavior we only have to annotate the ProductDao class with @Component:

```
@Component
public class ProductDao {
    //...
}

@Resource
public class ProductsController {

    private ProductDao dao;

    public ProductsController(ProductDao dao) {
        this.dao = dao;
    }

    public List<Product> list() {
        return dao.listAll();
    }

}
```

2.4- Add form: redirecting the request

We have a Products listing, but no way to register products. Thus, let's create a form for adding products. Since it is not a good idea to access the jsps directly, let's create an empty method that only redirects to a jsp:

```
@Resource
public class ProductsController {
```

```
    //...
    public void form() {
    }
}
```

So we can access the form by URI `/products/form`, and the form will be at `/WEB-INF/jsp/products/form.jsp`:

```
<form action="<c:url value="/products/adiciona"/>">
    Name:             <input type="text" name="product.name" /><br/>
    Description:<input type="text" name="product.description" /><br/>
    Price:            <input type="text" name="product.price" /><br/>
    <input type="submit" value="Save" />
</form>
```

This form will save a product using the URI `/products/add`, so we must create this method on the controller:

```
@Resource
public class ProductsController {
    //...
    public void add() {
    }
}
```

Look at the input names: **product.name**, **product.description** and **product.price**. If we receive a `Product` named `product` as parameter on add method, VRaptor will set the fields **name**, **description** and **price** with the input values, using the corresponding setters on Product. The **product.price** parameter will also be converted into Double before being set on the product. More information on Converters chapter.

Thus, having the correct names on the form inputs, we can create the add method:

```
@Resource
public class ProductsController {
    //...
    public void add(Product product) {
        dao.save(product);
    }
}
```

Right after saving something on a form we usually want to be redirected to the listing or back to the form. In this case we want to be redirected to the products listing. For this purpose there is a VRaptor component: the `Result`. It is responsible for adding attributes on the request, and for dispatching to a different view. To get a Result instance you must receive it as a constructor parameter:

```
@Resource
public class ProductsController {
    public ProductsController(ProductDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }
}
```

In order to redirect to the listing, you can use the result object:

```
result.use(Results.logic()).redirectTo(ProductsController.class).list();
```

This code snippet can be read as: *As a result, use a logic, redirecting to the list method in ProductsController*. All redirect configuration is 100% java code, with no strings involved! It's clear from the code that the result from your logic is not the default, and which one you're using. There is no need to worry about configuration files. Furthermore, if you need to rename the `list` method, there is no need to go through your entire application looking for redirects to this method, just use your usual refactoring IDE to do the rename.

Our `add` method would look like this:

```
public void add(Product product) {
    dao.add(product);
    result.use(Results.logic()).redirectTo(ProductsController.class).list();
}
```

You can get more info on `Result` at the Views and Ajax chapter.

2.5- Validation

It wouldn't make sense adding a nameless product in the system, nor a negative value for it's price. Before adding the product, we need to check if it is a valid product - which has a name and a positive price. In case it's not valid, we want to get back to the form and show error messages.

In order to do that, we can use a VRaptor component: the Validator. You can get it in your Controller's constructor and use it like this:

```
@Resource
public class ProductsController {
    public ProductsController(ProductDao dao, Result result, Validator validator) {
        //...
        this.validator = validator;
    }

    public void add(Product product) {
        validator.checking(new Validations() {{
            that(!product.getName().isEmpty(), "product.name", "nome.empty");
            that(product.getPrice() > 0, "product.price", "price.invalid");
        }});
        validator.onErrorUse(Results.page()).of(ProductsController.class).form();

        dao.add(product);
        result.use(Results.logic()).redirectTo(ProductsController.class).list();
    }
}
```

we can read the validation code as *Validate that the name of the product is not empty and that the product's price is bigger than zero. If an error occur, use the ProductsController form page as the result*. Therefore, if the product name is empty, the "name.empty" internationalized message will be added to the "product.name" field. If any error occurs, the system will get the user back to the form page, with all fields set, and error messages that can be accessed like this:

```
<c:forEach var="error" items="${errors}">
```

```
    ${error.category}  ${error.message}<br />
</c:forEach>
```

More information on Validation on, well, Validations chapter.

If you learnt what we said so far, you're able to make 90% of your application. Next sessions on this tutorial show the solution for some of the most frequent problems that lay on that 10% left.

[sectin Using Hibernate to store Products]

Let's make a real implementation of ProductDao, now, using Hibernate to persist products. You'll need a Session in your ProductDao. Using injection of dependencies, you'll have to declare you'll receive a Session in your constructor.

```
@Component
public class ProductDao {

    private Session session;

    public ProductDao(Session session) {
        this.session = session;
    }

    public void add(Product product) {
        session.save(product);
    }
    //...
}
```

But, wait, for VRaptor to know how to create that Session, and I can't simply put a @Component on the Session class because it is a Hibernate class. That's the reason why the ComponentFactory interface was created. More info on creating your own ComponentFactories can be found in Components chapter. You can also use the ComponentFactories available in VRaptor, as shown in the Utils chapter.

2.6- Controlling transactions: Interceptors

We often want to intercept as requests (or some of them) and execute a business logic, such as in a transaction control. That why VRaptor has interceptors. Learn more about them on the Interceptors' chapter. There is also an implemented TransactionInterceptor in VRaptor - learn how to use it on the Utils chapter.

2.7- Shopping Cart: session components

If we want to make a shopping cart in our system, we need some way to keep cart items in the user's session. In order to do it, we can create a session scoped component, i.e., a component that will last as long as the user session last. For that, simply create a component and annotate it with @SessionScoped:

```
@Component
@SessionScoped
public class ShoppingCart {
    private List<Product> items = new ArrayList<Product>();

    public List<Product> getAllItems() {
        return items;
    }
```

```
    public void addItem(Product item) {
        items.add(item);
    }
}
```

As this shopping cart is a component, we can receive it on the shopping cart's Controller's constructor:

```
@Resource
public class ShoppingCartController {

    private final ShoppingCart cart;

    public ShoppingCartController(ShoppingCart cart) {
        this.cart = cart;
    }

    public void add(Product product) {
        cart.addItem(product);
    }

    public List<Product> listItems() {
        return cart.getAllItems();
    }
}
```

Besides session scope, there is also the application scope and the @ApplicationScoped annotation. Components annotated with @ApplicationScoped will be created only once for the whole application.

2.8- A bit of REST

On REST's ideal of URIs identifying resources on the web to make good use of the structural advantages the HTTP protocol provides us, observe how simple it is, in VRaptor, mapping the different HTTP methods in the same URI to invoke different Controllers' methods. Suppose we want to use the following URIs on the products' crud:

```
GET /products - lista todos os products
POST /products - adiciona um product
GET /products/{id} - visualiza o product com o id passado
PUT /products/{id} - atualiza as informações do product com o id passado
DELETE /products/{id} - remove o product com o id passado
```

In order to create a REST behaviour in VRaptor, we can use the @Path annotations - that changes the URI to access a given method. Also, we use the annotations that indicate which HTTP methods are allowed to call that logic - @Get, @Post, @Delete and @Put.

A REST version of our `ProductsControler` would be something like that:

```
public class ProductsController {
    //...

    @Get
    @Path("/products")
```

```
    public List<Product> list() {...}

    @Post
    @Path("/products")
    public void add(Product product) {...}

    @Get
    @Path("/products/{product.id}")
    public void view(Product product) {...}

    @Put
    @Path("/products/{product.id}")
    public void update(Product product) {...}

    @Delete
    @Path("/products/{product.id}")
    public void remove(Product product) {...}

}
```

Note we can receive parameters on the URIs. For instance, if we can the **GET /products/5** URI, the `view` method will be invoked and the `product` parameter will have its id set as `5`.

More info on that are on the REST Resources chapter.

2.9- Message bundle File

Internationalization (i18n) is a powerful feature present in almost all Web frameworks nowadays. And it's no different with VRaptor3. With i18n you can make your applications support several different languages (such as French, Portuguese, Spanish, English, etc) in a very easy way: simply translating the application messages.

In order to support i18n, you must create a file called `messages.properties` and make it available in your application classpath (`WEB-INF/classes`). That file contains lines which are a set of key/value entries, for example:

```
field.userName = Username
field.password = Password
```

So far, it's easy, but what if you want to create files containing messages in other languages, for example, Portuguese? Also easy. You just need to create another properties file called `messages_pt_BR.properties`. Notice the suffix *_pt_BR* on the file name. It indicates that when the user access your application from his computer configured with Brazilian Portuguese locale, the messages in this file will be used. The file contents would be:

```
field.userName = Nome do Usuário
field.password = Senha
```

Notice that the keys are the same in both files, what changes is the value to the specific language.

In order to use those messages in your JSP files, you could use JSTL. The code would go as follows:

```
<html>
    <body>
        <fmt:message key="field.userName" /> <input name="user.userName" />
```

```
        <br />

        <fmt:message key="field.password" /> <input type="password" name="user.password" />

        <input type="submit" />
    </body>
</html>
```

Resources-Rest

3.1- What are Resources?

Resources are anything that can be accessed by our clients.

In a VRaptor-based Web application, a resource must be annotated with `@Resource`. If you annotate a class with `@Resource`, all its public methods become accessible through GET requests to specific URIs.

The following example shows a resource called `ClienteController`, which provides several operations over clients.

Creating the class below with all its methods instantly make the URIs **/client/add**, **/client/list**, **/client/show**, **/client/remove** and **/client/update** available, each one invoking the respective method.

```
@Resource
public class ClientController {

  public void add(Client client) {
    ...
  }

  public List<Client> list() {
    return ...
  }

  public Client show(Client profile) {
    return ...
  }

  public void remove(Client client) {
    ...
  }

  public void update(Client client) {
    ...
  }

}
```

3.2- Parâmetros dos métodos

You can receive parameters on your controller methods, and if those parameters follow the java beans convention (getters and setters for class fields), you can use dots for browsing through the fields. For instance, on method:

```
public void update(Client client) {
   //...
```

```
}
```

you can receive on the request parameters:

```
client.id=3
client.name=John Doe
client.user.login=johndoe
```

and the respective fields will be set, browsing through getters and setters starting from client.

If an object field or a method parameter is a list (List<> or array), you can receive several request parameters, using square brackets and indexes:

```
client.phones[0]=+55 11 5571-2751 #if it is a string list
client.relatives[0].id=1 #if it is an arbitrary object, you can continue to browse
client.relatives[3].id=1 #indexes don't need to be sequential
client.relatives[0].name=Mary Doe #using the same index, it will be set on same object
clients[1].id=23 #it works if you receive a client list as method parameter
```

Reflection on parameter names

Unfortunately Java can't reflect parameters names, these data don't stay in bytecode (unless you compile your code in debug mode, but that is optional). It causes that most frameworks that need this kind of information end up creating annotations, which makes a very ugly code (like JAX-WS, where its very common to find methods with signature like: `void add(@WebParam(name="client") Client client)`).
VRaptor uses the Paranamer framework (http://paranamer.codehaus.org), which can get parameter names information through pre compilation or debug data, avoiding the creation of annotations for this purpose. Some VRaptor developers also participate in Paranamer development.

3.3- Scopes

Sometimes you want to share a component among all users, or through all requests from the same user or one instance for each user request.

To specify in which scope your component will live, use the annotations `@ApplicationScoped`, `@SessionScoped` and `@RequestScoped`.

If you don't specify a scope for your component, VRaptor assumes the request scope, meaning a fresh instance will be created for each request.

3.4- Http Methods

The best practice when using HTTP Methods is to specify a different URI for each method, like GET, POST, PUT etc.

In order to accomplish that, we use annotations `@Get`, `@Post`, `@Delete` etc, in conjunction with the `@Path` annotation, which allows us to configure a custom URI.

The following example changes the default URIs for `ClienteController`. Now we specify two different URIs for different HTTP methods:

```
@Resource
public class ClientController {

  @Path("/client")
  @Post
  public void add(Client client) {
  }

  @Path("/")
  public List<Client> list() {
    return ...
  }

  @Get
  @Path("/client")
  public Client show(Client client) {
    return ...
  }

  @Delete
  @Path("/client")
  public void remove(Client client) {
    ...
  }

  @Put
  @Path("/client")
  public void update(Client client) {
    ...
  }

}
```

As you can see, we used HTTP methods + a specific URI to identify each method in our Java class.

We must be **very careful** when creating hyperlinks and HTML forms, because web browsers currently support only *POST* and *GET* methods.

For that reason, requests for methods *DELETE*, *PUT* etc should be created through JavaScript, or by adding an extra parameter called **_method**.

This parameter will overwrite the real HTTP method being invoked.

The following example creates a link to show one client's data:

```
<a href="/client?client.id=5">show client 5</a>
```

Now an example on how to invoke the method to add a client:

```
<form action="/client" method="post">
    <input name="client.name" />
    <input type="submit" />
</form>
```

Notice that if we want to remove a cliente using the *DELETE* HTTP method, we have to use the `_method` parameter, since browsers still don't support that kind of requests:

```
<form action="/client" method="post">
    <input name="_method" value="DELETE" type="hidden" />
```

```
    <input name="client.id" value="5" type="hidden" />
    <input type="remove client 5" />
</form>
```

3.5- @Path

The `@Path` annotation allows you to specify custom access URIs to your controller methods. The basic usage of the annotation is to specify a fixed URI. The following example shows how to customize the access URI for a method that accepts *POST* requests only. The URI we want to specify is /**client**:

```
@Resource
public class ClientController {

  @Path("/client")
  @Post
  public void add(Client client) {
  }

}
```

Path with variable injection

Sometimes we want the *uri* to include, for example, the unique identifier of my resource.

Suppose a client controller application where the client's unique identifier (primary key) is a number. We can map our URIs as */client/{client.id}*, so we can visualize each client.

That is, if we access the URI */client/2*, the **show** method will be invoked and the *client.id* parameter will be set to **2**. If the URI */client/1717* is accessed, the same method will be invoked with the **1717** value.

That way we can create unique URIs to identify different resources in our application. See the mentioned example:

```
@Resource
public class ClientController {

  @Get
  @Path("/client/{client.id}")
  public Cliente show(Client client) {
    return ...
  }

}
```

You can go further and set several parameters through the URI:

```
@Resource
public class ClientController {

  @Get
  @Path("/client/{client.id}/show/{section}")
  public Client show(Client client, String section) {
    return ...
  }
```

```
}
```

Paths with wildcards

You can also use the * wildcard as a selection method for your URI. The following example ignores anything that comes after the word *photo/* :

```
@Resource
public class ClientController {

  @Get
  @Path("/client/{client.id}/photo/*")
  public File photo(Client client) {
    return ...
  }

}
```

And now a similar code, but used to download a specific photo from a client:

```
@Resource
public class ClientController {

  @Get
  @Path("/client/{client.id}/photo/{photo.id}")
  public File photo(Client client, Photo photo) {
    return ...
  }

}
```

Sometimes you want the parameter to include the / character. In that case, you should use the pattern {...*}:

```
@Resource
public class ClientController {

  @Get
  @Path("/client/{client.id}/download/{path*}")
  public File download(Client client, String path) {
    return ...
  }

}
```

Specifying priorities for your paths

It is possible for some URIs to be handled by more than one method in our class:

```
@Resource
public class PostController {
```

```
  @Get
  @Path("/post/{post.author}")
  public void show(Post post) { ... }

  @Get
  @Path("/post/current")
  public void current() { ... }
}
```

The URI `/post/current` can be handled by both `show` and `current` methods. But I don't want to invoke the `show` method with that URI, what I want is VRaptor to test the `current` path first, avoiding the invocation of the `show` method.

In order to do that, we can define priorities for `@Path`s, so VRaptor will first test paths with higher priority, in other words, paths with lower priority values.

```
@Resource
public class PostController {

  @Get
  @Path(priority = 2, value = "/post/{post.author}")
  public void show(Post post) { ... }

  @Get
  @Path(priority = 1, value = "/post/current")
  public void current() { ... }
}
```

This way, the "/post/current" path will be tested before "/post/{post.author}" by VRaptor, solving our problem.

3.6- RoutesConfiguration

Finally, the most advanced way to configure access routes for your resources is using a **RoutesConfiguration**.

This component must be configured as application scoped and must implement the *config* method:

```
@Component
@ApplicationScoped
public class CustomRoutes implements RoutesConfiguration {

    public void config(Router router) {
    }

}
```

Having access to a **Router**, you can define access routes to methods. And the best part is that the configuration is refactor-friendly, that is, if you change a method's name, the configuration reflects the change, but the *uri* stays the same:

```
@Component
@ApplicationScoped
public class CustomRoutes implements RoutesConfiguration {
```

```
    public void config(Router router) {
        new Rules(router) {
            public void routes() {
                routeFor("/").is(ClientController.class).list();
                routeFor("/client/random").is(ClientController.class).random();
            }
        };
    }

}
```

You can also put parameters on the uri and they will be set directly on the method parameters. You can also add restrictions to these parameters:

```
// show method receives a Client that has an id
routeFor("/client/{client.id}").is(ClientController.class).show(null);
// If I want to ensure that the parameter is a number:
routeFor("/client/{client.id}").withParameter("client.id").matching("\\d+")
            .is(ClientController.class).show(null);
```

At last, you can choose the class and the method names in runtime, allowing us to create extremely generic routes:

```
routeFor("/{webResource}/doSomething/{webMethod}").is(
    type("br.com.caelum.projectname.{webResource}"),
    method("{webMethod}"));
```

Components

4.1- What are components?

Components are object instances that your application need to execute tasks or to keep state in different situations.

DAOs and e-mail senders are classic component examples.

The best practices suggest you should *always* create interfaces for your components to implement. This makes your code much easier to unit test.

The following example shows a VRaptor-managed component:

```
@Component
public class ClientDao {

  private final Session session;
  public ClientDao(HibernateControl control) {
      this.session = control.getSession()
  }

  public void add(Client client) {
    session.save(client);
  }

}
```

4.2- Scopes

Just like resources, components live in specific scopes and follow the same rules. The default scope for a component is the request scope, meaning that a new instance will be created for each request.

The following example shows a Hibernate-based connection provider. The application scope is specified for the provider, so only one instance per application context will be created:

```
@ApplicationScoped
@Component
public class HibernateControl {

  private final SessionFactory factory;
  public HibernateControl() {
      this.factory = new AnnotationConfiguration().configure().buildSessionFactory();
  }

  public Session getSession() {
      return factory.openSession();
  }
```

```
}
```

4.3- ComponentFactory

It can happen that one of your class dependencies doesn't belong to your project, like the Session from Hibernate or EntityManager from JPA.

In order to do that you can create a ComponentFactory:

```
@Component
public class SessionCreator implements ComponentFactory<Session> {

    private final SessionFactory factory;
    private Session session;

    public SessionCreator(SessionFactory factory) {
        this.factory = factory;
    }

    @PostConstruct
    public void create() {
        this.session = factory.openSession();
    }

    public Session getInstance() {
        return session;
    }

    @PreDestroy
    public void destroy() {
        this.session.close();
    }

}
```

Note that you can add listeners like @PostConstruct and @PreDestroy to manage creation and destruction of you factory resources. You can use these listeners on any component that you register on VRaptor.

4.4- Dependency injection

VRaptor uses one of its own dependency injection providers to control what it needs in order to create new instances of your components and resources.

For that reason, the former two examples allow any of your resources or components to receive a ClientDao in its constructor. For example:

```
@Resource
public class ClientController {
    private final ClientDao dao;

    public ClientController(ClientDao dao) {
        this.dao = dao;
    }
```

```
    @Post
    public void add(Client client) {
        this.dao.add(client);
    }

}
```

Converters

5.1- Default

VRaptor registers a default set of converters for your day-to-day use.

5.2- Primitive types

All primitive types (int, long etc) are supported.

If the request parameter is empty or null, primitive type variables will be set to its default value, as if it was a class attribute. In general:

- boolean - false
- short, int, long, double, float, byte - 0
- char - caracter de código 0

5.3- Primitive type wrappers

All primitive type wrappers (Integer, Long, Character, Boolean etc) are supported.

5.4- Enum

Enums are also supported using the element's name or ordinal value. In the following example, either 1 or DEBIT values are converted to Type.DEBIT:

```
public enum Type {
    CREDIT, DEBIT
}
```

5.5- BigInteger and BigDecimal

Both are supported using your JVM's default locale. To enable decimal values based on the user's locale, you can check how the class LocaleBasedCalendarConverter works.

5.6- Calendar and Date

Both LocaleBasedCalendarConverter and LocaleBasedDateConverter are based on the user's locale, defined using JSTL pattern to understand the parameter's format.

For example, if the locale is pt-br, then "18/09/1981" stands for September 18th 1981. On the other hand, if the locale is en, the same date is formatted as "09/18/1981".

5.7- Interface

All converters must implement VRaptor's Converter interface. The concrete class will define which type it is able to convert, and will be invoked with a request parameter, the target type and a resource bundle containing i18n messages, useful if you wish to raise a ConversionException in case of convertion errors.

```
public interface Converter<T> {
    T convert(String value, Class<? extends T> type, ResourceBundle bundle);
}
```

Also, your must tell VRaptor (not the compiler) which type your converter is able to handle. You do that by annotating your converter class with @Convert:

```
@Convert(Long.class)
public class LongConverter implements Converter<Long> {
    // ...
}
```

Finally, don't forget to specify the scope of your converter, just like you do with any other resource in VRaptor. For example, if your converter doesn't need any user specific information, it can be registered as application scoped and only one instance of that converter will be created:

```
@Convert(Long.class)
@ApplicationScoped
public class LongConverter implements Converter<Long> {
    // ...
}
```

In the following lines, you can see a LongConverter implementation, showing how simple it is to assemble all the information mentioned above:

```
@Convert(Long.class)
@ApplicationScoped
public class LongConverter implements Converter<Long> {

    public Long convert(String value, Class<? extends Long> type, ResourceBundle bundle) {
        if (value == null || value.equals("")) {
            return null;
        }
        try {
            return Long.valueOf(value);
        } catch (NumberFormatException e) {
            throw new
    ConversionError(MessageFormat.format(bundle.getString("is_not_a_valid_integer"), value));
        }
    }

}
```

5.8- Registering a new converter

No further configuration is needed except implementing the Converter interface and annotating the converter class with @Convert for your custom converter to be registered in VRaptor's container.

5.9- More complex converters

Interceptors

6.1- Why intercept

Interceptors are implemented in order to execute tasks before and/or after the execution of a business logic, being data validation, database connection and transaction control, logging and data cryptography/compression the most common use cases.

6.2- How to intercept

In VRaptor 3 we adopted an approach in which the interceptor defines who will be intercepted. This is closer to the intercepting style used in systems based on AOP (Aspect Oriented Programming) than the one that was implemented in VRaptor's previous version.

Therefore, to intercept a request, just implement the **Interceptor** interface and annotate the class with **@Intercepts**.

Just like any other component, you can specify the interceptor's scope using the scope annotations.

```
public interface Interceptor {

    void intercept(InterceptorStack stack, ResourceMethod method,
                   Object resourceInstance) throws InterceptionException;

    boolean accepts(ResourceMethod method);

}
```

6.3- Simple example

The following class shows an example of how to intercept all requests using session scope and simply print the invocation to default output.

Remember that the interceptor is a component just like any other, so it can receive its dependencies in the constructor through Dependency Injection.

```
@Intercepts
@SessionScoped
public class Log implements Interceptor {

    private final HttpServletRequest request;

    public Log(HttpServletRequest request) {
        this.request = request;
    }
```

```
    public void intercept(InterceptorStack stack, ResourceMethod method,
                          Object resourceInstance) throws InterceptionException {
        System.out.println("Intercepting " + request.getRequestURI());
        stack.next(method, resourceInstance);
    }

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

}
```

6.4- Example using Hibernate

Probably one of the most common uses of an Interceptor is to implement the Open Session In View pattern, which provides a database connection whenever a request is made to your application. And in the end of that request, the connection is disposed. This is specially useful to avoid exceptions like LazyInitializationException when rendering JSPs.

Here is a simple example that starts a database transaction in every request, and when the logic execution ends and the page is rendered, it commits the transaction and closes the database connection.

```
@RequestScoped
@Intercepts
public class DatabaseInterceptor implements br.com.caelum.vraptor.Interceptor {

    private final Database controller;
    private final Result result;
    private final HttpServletRequest request;

    public DatabaseInterceptor(Database controller, Result result, HttpServletRequest request) {
        this.controller = controller;
        this.result = result;
        this.request = request;
    }

    public void intercept(InterceptorStack stack, ResourceMethod method,
                          Object instance) throws InterceptionException {
        result.include("contextPath", request.getContextPath());
        try {
            controller.beginTransaction();
            stack.next(method, instance);
            controller.commit();
        } finally {
            if (controller.hasTransaction()) {
                controller.rollback();
            }
            controller.close();
        }
    }

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

}
```

This way, to use the available connection in your Resource, the following code would apply:

```
@Resource
public class EmployeeController {

    public EmployeeController(Result result, Database controller) {
        this.result = result;
        this.controller = controller;
    }

    @Post
    @Path("/employee")
    public void add(Employee employee) {
        controller.getEmployeeDao().add(employee);
        ...
    }
}
```

6.5- How to ensure ordering: InterceptorSequence

If you want to ensure order of the execution of a set of interceptors, you can implement the interface InterceptorSequence and return the order you want to execute the interceptors:

```
@Intercepts
public class MySequence implements InterceptorSequence {
    public Class<? extends Interceptor>[] getSequence() {
        return new Class[] { FirstInterceptor.class, SecondInterceptor.class };
    }
}
```

You should not annotate the interceptors returned by InterceptorSequence.getSequence() with @Intercepts.

Validation

VRaptor3 supports two different validation styles: classic and fluent. The starting point to both styles is the Validator interface. In order to access the Validator, your resource must receive it in the constructor:

```
import br.com.caelum.vraptor.Validator;

...

@Resource
class EmployeeController {
    private Validator validator;

    public EmployeeController(Validator validator) {
        this.validator = validator;
    }
}
```

7.1- Classic style

The classic style is very similar to VRaptor2's validation. Inside your business logic, all you have to do is check the data you want, and if you find any validation errors, add them to the errors list. For example, to validate that employee name is 'John Doe':

```
public void add(Employee employee) {
    if (!employee.getName().equals("John Doe")) {
        validator.add(new ValidationMessage("error","invalidName"));
    }
    validator.onErrorUse(page()).of(EmployeeController.class).form();

    dao.add(employee);
}
```

When you call validator.onErrorUse, if there are any validation errors, VRaptor will stop execution and redirect to the page you specified. This redirect has the same behavior as the result.use(..) redirects.

7.2- Fluent style

The goal of fluent style is to write the validation code in such way that it feels natural. For example, if we want the employee name to be required:

```
public add(Employee employee) {
    validator.checking(new Validations(){{
        that(!employee.getName().isEmpty(), "error","nameIsRequired");
    }});
```

```
    validator.onErrorUse(page()).of(EmployeeController.class).form();

    dao.add(employee);
}
```

You can read the code above like this: “Validator, check my validations. First one is that employee name cannot be empty”. Much closer to natural language.

So, if employee name is empty, the flow will be redirected to the “form” logic, which shows the user a form to insert employee data again. Also, the error message is sent back to the form.

There are validations that may occur only if other validation succeeded, for instance I will check user age only if the user is not null. The that method will return a boolean that represents the success of the validation:

```
validator.checking(new Validations(){{
    if (that(user != null, "user", "null.user")) {
        that(user.getAge() >= 18, "user.age", "user.is.underage");
    }
}});
```

So the second validation will execute only if the first didn’t fail.

7.3- Validation using Hamcrest Matchers

You can use Hamcrest matchers for making validation even more fluent and readable, with the advantage of matcher composition and the creation of new matchers that Hamcrest allows:

```
public admin(Employee employee) {
    validator.checking(new Validations(){{
        that(employee.getRoles(), hasItem("ADMIN"), "admin","employee.is.not.admin");
    }});
    validator.onErrorUse(page()).of(LoginController.class).login();
    dao.add(employee);
}
```

7.4- Hibernate validator

VRaptor 3 also supports HibernateValidator integration. In the example above, to validate the employee object using HibernateValidator, just add one line to your code:

```
public add(Employee employee) {
    //Validation with Hibernate Validator
    validator.add(Hibernate.validate(employee));

    validator.checking(new Validations(){{
        that(!employee.getName().isEmpty(), "error","nameIsRequired");
    }});
    validator.onErrorUse(page()).of(EmployeeController.class).form();

    dao.add(employee);
}
```

7.5- Where to redirect in case of errors

Another issue that one must consider when validating data is where to redirect when an error occurs. How do one redirect the user to another resource using VRaptor3 in case of validation errors?

Easy, just tell your validator to do just that: when you find any validation error, send the user to the specified resource. See the example:

```
public add(Employee employee) {

    //Fluent validation
    validator.checking(new Validations(){{
        that(!employee.getName().isEmpty(), "error","nameIsRequired");
    }});

    //Classic validation
    if (!employee.getName().equals("John Doe")) {
        validator.add(new ValidationMessage("error","invalidName"));
    }
    validator.onErrorUse(page()).of(EmployeeController.class).form();

    dao.add(employee);
}
```

If your logic may add any validation error you **must** specify where to go in case of error. Validator.onErrorUse works just like result.use: you can use any view from Results class.

View and Ajax

8.1- Custom PathResolver

By default, VRaptor tries to render your views following the convention:

```
public class ClientsController {
    public void list() {
        //...
    }
}
```

The method listed above will render the view `/WEB-INF/jsp/clients/list.jsp`.

However, we don't always want it to behave that way, specially if we need to use some template engine like Freemarker or Velocity. In that case, we need to change the convention.

An easy way of changing that convention is extending the `DefaultPathResolver` class:

```
@Component
public class FreemarkerPathResolver extends DefaultPathResolver {
    protected String getPrefix() {
        return "/WEB-INF/freemarker/";
    }

    protected String getExtension() {
        return "ftl";
    }
}
```

That way, the logic would try to render the view `/WEB-INF/freemarker/clients/list.ftl`. If that solution is not enough, you can implement the `PathResolver` interface and do whatever convention you wish. Don't forget to annotate your new classe with `@Component`.

8.2- View

If you want to change a specific logic's view, you can use the `Result` object:

```
@Resource
public class ClientsController {

    private final Result result;

    public ClientsController(Result result) {
        this.result = result;
    }
```

```
    public void list() {}

    public void save(Client client) {
        //...
        this.result.use(Results.logic()).redirectTo(ClientsController.class).list();
    }
}
```

By default, there are these view implementations:

- Results.logic(), redirects to any other logic in the application.
- Results.page(), redirects directly to a page, that can be a jsp, an html, or any URI relative to the web application directory or the application context.
- Results.http(), sends HTTP protocol informations, like status codes and headers.
- Results.referer(), uses Referer header to redirect or forward.
- Results.nothing(), simply returns the HTTP success code (HTTP 200 OK).

8.3- Redirect and forward

In VRaptor3, you can either redirect or forward the user to another logic or page. The main difference between redirecting and forwarding is that the former happens at client side, while the latter happens at server side.

A good redirect use is the pattern 'redirect-after-post', for example, when you add a client and you want to return to the client listing page, but you want to avoid the user to accidentally resend all data by refreshing (F5) the page.

An example of forwarding is when you have some data validation that fails, usually you want the user to remain on the form with all the previously filled data.

Automatic Flash Scope

If you add objects on Result and redirects to another logic, these objects will be available on the next request.

```
public void add(Client client) {

    dao.add(client);
    result.include(Automatic Flash Scope"noticeAutomatic Flash Scope", Automatic Flash
Scope"Client successfully addedAutomatic Flash Scope");
    result.use(logic()).redirectTo(ClientsController.class).list();
}
```

list.jsp:

```
...
Automatic Flash Scope<div idAutomatic Flash Scope=Automatic Flash
Scope"noticeAutomatic Flash Scope"Automatic Flash Scope>
   Automatic Flash Scope<h3Automatic Flash Scope>${notice}Automatic Flash
Scope<Automatic Flash Scope/h3Automatic Flash Scope>
Automatic Flash Scope<Automatic Flash Scope/divAutomatic Flash Scope>
...
```

8.4- Accepts and the _format parameter

Many times you need to render different formats for the same logic. For example, we want to return a JSON object instead of an HTML page. In order to do that, we can define the request's Accepts header to accept the desired format, or we can pass a `_format` paramenter in the request.

If the specified format is JSON, the default rendered view will be: `/WEB-INF/jsp/{controller}/{logic}.json.jsp`, which means, in general, the rendered view will be: `/WEB-INF/jsp/{controller}/{logic}.{format}.jsp`. If the format is HTML, then you won't need to specify it in the file name.

The `_format` parameter has a higher priority over the Accepts header.

8.5- Ajax: building on the view

In order to return a JSON object to the view, your logic must make that object available somehow. Just like the following example, your `/WEB-INF/jsp/clients/load.json.jsp`:

```
{ name: '${client.name}', id: '${client.id}' }
```

And in the controller:

```
@Resource
public class ClientsController {

    private final Result result;
    private final ClientDao dao;

    public ClientsController(Result result, ClientDao dao) {
        this.result = result;
        this.dao = dao;
    }

    public void load(Client client) {
        result.include("client", dao.load(client));
    }
}
```

Dependency injection

VRaptor is strongly based on Dependency Injection, since all its internal components are managed using this technique.

Tha basic concept behind Dependency Injection (DI) says you should not look for what you want to access. Instead, it should be provided for you somehow.

In Java, this is accomplished by passing components to your controller's constructor. Suppose your clients controller needs to access a clients Dao. Specify that need in your code:

```
@Component

public class ClientController {
    private final ClientDao dao;

    public ClientController(ClientDao dao) {
        this.dao = dao;
    }

    @Post
    public void add(Client client) {
        this.dao.add(client);
    }

}
```

And annotate the ClientDao component as a VRaptor `@Component`:

```
@Component

public class ClientDao {
}
```

From now on, VRaptor will provide your `ClientController` with an instance of `ClientDao` when needed. Remember that VRaptor will honor the scope specified by the component. For example, if `ClientDao` had specified Session scope (`@SessionScoped`), only one instance of that component would be created per session. (Note that it is probably wrong to specify session scope for a Dao, it is only a simple example).

9.1- ComponentFactory

Sometimes we want our components to receive other libraries' components. In that case we are unable to change the libraries's source code in order to annotate its components with `@Component` (and any other changes we may need to do).

The most common example is acquiring a Hibernate Session. We need to create a component that is responsible for providing Session instances for other components that depend on it.

VRaptor has an interface called `ComponentFactory` which allows your classes to provide components.

Classes implementing that interface define a single method. See the following example, which starts Hibernate when the component is built and uses that configuration to provide Session instances for our application:

```
@Component
@ApplicationScoped
public class SessionFactoryCreator implements ComponentFactory<SessionFactory> {

    private SessionFactory factory;

    @PostConstruct
    public void create() {
        factory = new AnnotationConfiguration().configure();
    }

    public SessionFactory getInstance() {
        return factory;
    }

    @PreDestroy
    public void destroy() {
        factory.close();
    }

}

@Component
@RequestScoped
public class SessionCreator implements ComponentFactory<Session> {

    private final SessionFactory factory;
    private Session session;

    public SessionCreator(SessionFactory factory) {
        this.factory = factory;
    }

    @PostConstruct
    public void create() {
        this.session = factory.openSession();
    }

    public Session getInstance() {
        return session;
    }

    @PreDestroy
    public void destroy() {
        this.session.close();
    }


}
```

These implementations are already on VRaptor3. Utils chapter will show you how to use them.

9.2- Providers

Behind the curtains, VRaptor uses a specific DI provider and has out-of-the-box support for PicoContainer or Spring DI.

Each implementation give you all you can find in VRaptor's documentation, but also different extension points.

9.3- Spring

When using Spring, you gain all its features and built-in components to use with VRaptor. In other words, all components that work with Sprint DI/IoC, also work with VRaptor. In that case, all the annotations.

You don't have to configure anything, since Spring is the default container.

VRaptor will use your Spring configurations, if you have it already configured in your project (Context listeners and applicationContext.xml). If VRaptor can't find your Spring configuration you can configure it, as you can see on Advanced Configurations Chapter.

9.4- Pico Container

When using PicoContainer with VRaptor, you will be able to access Pico directly in order to do advanced configurations.

If you want PicoContainer to be your application's DI provider, put the following entries in your web.xml descriptor:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.caelum.vraptor.ioc.pico.PicoProvider</param-value>
</context-param>
```

9.5- Your custom provider

You can also create your own Provider, either to extend the default implementations with PicoContainer or Spring, or to base your implementation in another DI container you may prefer.

Downloading

10.1- 1 minute example: download

The following example shows how to expose the file to be downloaded to its client.

Again, see how simple this code is:

```
@Resource
public class ProfileController {

    public File picture(Profile profile) {
        return new File("/path/to/the/picture." + profile.getId()+ ".jpg");
    }
}
```

10.2- Adding more info to download

If you want to add more information to download, you can return a FileDownload:

```
@Resource
public class ProfileController {

    public Download picture(Profile profile) {
        File file = new File("/path/to/the/picture." + profile.getId()+ ".jpg");
        String contentType = "image/jpg";
        String filename = profile.getName() + ".jpg";

        return new FileDownload(file, contentType, filename);
    }
}
```

10.3- 1 minute example: upload

The first example is based on the multipart upload feature.

```
@Resource
public class ProfileController {

    private final ProfileDao dao;

    public ProfileController(ProfileDao dao) {
        this.dao = dao;
    }
```

```
    public void updatePicture(Profile profile, UploadedFile picture) {
        dao.update(picture.getFile(), profile);
    }
}
```

10.4- More about Upload

UploadedFile returns the file content as a InputStream. If you want to save this file on disk in an easy way, you can use the `commons-io IOUtils`, that is already a VRaptor dependency:

```
public void updatePicture(Profile profile, UploadedFile picture) {
    File pictureOnDisk = new File();
    IOUtils.copy(picture.getFile(), new PrintWriter(pictureOnDisk));
    dao.atribui(pictureOnDisk, profile);
}
```

Utility Components

11.1- Registering optional components

VRaptor have some optional components, on package br.com.caelum.vraptor.util. For registering them you can do as follows:

- Create a child class of your DI Profile (Spring is the default):

```
package com.companyname.projectName;


public class CustomProvider extends SpringProvider {

}
```

- Register this class as your DI provider on web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>com.companyname.projectName.CustomProvider</param-value>
</context-param>
```

- Override the registerCustomComponents method and add your optional components:

```
package com.companyname.projectName;


public class CustomProvider extends SpringProvider {

    @Override
    protected void registerCustomComponents(ComponentRegistry registry) {
        registry.register(OptionalComponent.class, OptionalComponent.class);
    }
}
```

11.2- Hibernate Session and SessionFactory

If your components need Hibernate Session and SessionFactory, you will need a ComponentFactory to create them for you. If you use annotated entities, and you have a hibernate.cfg.xml in the root of WEB-INF/classes, you can use VRaptor's built-in ComponentFactory. All you have to do is:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(SessionCreator.class, SessionCreator.class);
    registry.register(SessionFactoryCreator.class, SessionFactoryCreator.class);
```

```
}
```

You can also enable the interceptor that manages Hibernate transactions:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.registry(HibernateTransactionInterceptor.class,
                        HibernateTransactionInterceptor.class);
}
```

11.3- JPA EntityManager e EntityManagerFactory

If you have a persistence.xml with the persistence-unit called “default”, you can use VRaptor3 built-in ComponentFactories for EntityManager and EntityManagerFactory:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(EntityManagerCreator.class, EntityManagerCreator.class);
    registry.register(EntityManagerFactoryCreator.class,
                        EntityManagerFactoryCreator.class);
}
```

You can also enable the interceptor that manages JPA transactions:

```
@Override
protected void registerCustomComponents(ComponentRegistry registry) {
    registry.register(JPATransactionInterceptor.class,
                        JPATransactionInterceptor.class);
}
```

Advanced configurations: overriding VRaptor's behavior and conventions

12.1- Changing the default rendered view

If you need to change the default rendered view, or change the place where it'll be look for, you'll only need to create the following class:

```
@Component
public class CustomPathResolver extends DefaultPathResolver {

    @Override
    protected String getPrefix() {
        return "/root/directory/";
    }

    @Override
    protected String getExtension() {
        return "ftl"; // or any other extension
    }

    @Override
    protected String extractControllerFromName(String baseName) {
        return //your convention here
               //ex.: If you want to redirect UserController to 'userResource' instead of 'user'
               //ex.2: If you override the convention for Controllers name to XXXResource
        //and still want to redirect to 'user' and not to 'userResource'
    }
}
```

If you need a more complex convention, just implement the PathResolver interface.

12.2- Mudando a URI padrão

The default URI for ClientsController.list() is /clients/list, i.e, controller_name/method_name. If you want to override this convention, you can create a class like:

```
@Component
@ApplicationScoped
public class MyRoutesParser extends PathAnnotationRoutesParser {
    //delegate constructor
    protected String extractControllerNameFrom(Class<?> type) {
        return //your convention here
    }

     protected String defaultUriFor(String controllerName, String methodName) {
```

```
        return //your convention here
    }
}
```

If you need a more complex convention, just implement the RoutesParser interface.

12.3- Changing IoC provider

VRaptor's default IoC provider is Spring. In order to change, just add the following to your web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>com.package.from.your.PrefferedProviderClass</param-value>
</context-param>
```

VRaptor comes with built-in support to both Spring (br.com.caelum.vraptor.ioc.spring.SpringProvider) and PicoContainer (br.com.caelum.vraptor.ioc.pico.PicoProvider). You still have the option to extend any of these classes and use your own provider.

12.4- Changing Spring's base ApplicationContext

If VRaptor isn't using your ApplicationContext as base, just extend SpringProvider and implement the getParentApplicationContext method, giving it your application's ApplicationContext:

```
package br.com.apackage.aproject;
public class CustomProvider extends SpringProvider {
    public ApplicationContext getParentApplicationContext(ServletContext context) {
        ApplicationContext applicationContext = //your own logic to create your applicationContext
        return applicationContext;
    }
}
```

and change the provider at your web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.apackage.aproject.CustomProvider</param-value>
</context-param>
```

By default, VRaptor try to find the applicationContext via `WebApplicationContextUtils.getWebApplicationContext(`
or loading the applicationContext.xml that is on your classpath.

12.5- Changing the application character encoding

For using an arbitrary character encoding on all your requests and responses, avoiding encoding inconsistencies, you can set this parameter on your web.xml.

```
<context-param>
    <param-name>br.com.caelum.vraptor.encoding</param-name>
    <param-value>UTF-8</param-value>
</context-param>
```

This way all of your pages and form data will use the UTF-8.

Testing components and controllers

VRaptor3 manages your class dependencies, so there is no need to worry about instantiating your components and controllers, you can just receive your dependencies on the constructor and VRaptor3 will locate them and instantiate your class.

You can take advantage of dependency injection when testing your classes: you can instantiate your class with fake implementations and unit test the class.

Nevertheless, there are two VRaptor3 components that are dependencies of most of your controllers: `Result` and `Validator`. Their fluent interfaces makes it difficult to create fake implementations or mocks. Therefore there are fake implementations for these components on VRaptor3: `MockResult` e `MockValidator`.

13.1- MockResult

MockResult ignores all redirects, and stores the included objects, so you can inspect them and make assertions.

This snippet shows you how you can use MockResult:

```
MockResult result = new MockResult();
ClientController controller = new ClientController(..., result);
controller.list(); // will call result.include("clients", something);
List<Client> clients = result.included("clients"); // the cast is implicit
Assert.assertNotNull(clients);
// more assertions
```

Any calls to result.use(...) will be ignored.

13.2- MockValidator

MockValidator will store generated errors, so if there is any error when validator.onErrorUse is called, a ValidationError will be thrown. Therefore you can inspect the added errors, or simply check if there is any error.

```
@Test(expected=ValidationException.class)
public void testThatAValidationErrorOccurs() {
    ClientController controller = new ClientController(..., new MockValidator());
    controller.add(new Client());
}
```

ou

```
@Test
public void testThatAValidationErrorOccurs() {
    ClientController controller = new ClientController(..., new MockValidator());
```

```
    try {
        controller.add(new Cliente());
        Assert.fail();
    } catch (ValidationException e) {
        List<Message> errors = e.getErrors();
        //assertions on errors
    }
}
```

ChangeLog

14.1- 3.0.1 (to be released))

- paranamer upgraded to version 1.5 (Update your jar!)
- jars split in optional and mandatory on vraptor-core
- dependencies are now explained on vraptor-core/libs/mandatory/dependencies.txt and vraptor-core/libs/optional/dependencies.txt
- you can set now your application character encoding on web.xml through the context-param br.com.caelum.vraptor.encoding
- new view: Referer view: result.use(Results.referer()).redirect();
- Flash scope:

```
result.include("aKey", anObject);

result.use(logic()).redirectTo(AController.class).aMethod();
```

  objects included on Result will survive until next request when a redirect happens.
- @Path annotation supports multiple values (String -> String[])
- Result.include returns this to enable a fluent interface (result.include(...).include(...))
- Better exception message when there is no such http method as requested
- FileDownload registers content-length
- Solving issue 117: exposing null when null returned (was exposing “ok”)
- Solving issue 109: if you have a file `/path/index.jsp`, you can access it now through `/path/`, unless you have a controller that handles this URI.
- When there is a route that can handle the request URI, but doesn’t allow the requested HTTP method, VRaptor will send a 405 -> Method Not Allowed HTTP status code, instead of 404.
- A big refactoring on Routes internal API.

14.2- 3.0.0

- ValidationError renamed to ValidationException
- result.use(Results.http()) for setting headers and status codes of HTTP protocol
- bug fixes

- documentation
- new site

14.3- 3.0.0-rc-1

- example application: mydvds
- new way to add options components into VRaptor:

```
public class CustomProvider extends SpringProvider {


    @Override
    protected void registerCustomComponents(ComponentRegistry registry) {
        registry.registry(OptionComponent.class, OptionComponent.class);
    }
}
```

- Utils: HibernateTransactionInterceptor and JPATransactionInterceptor
- Full application example inside the docs
- English docs

14.4- 3.0.0-beta-5

- New way to do validations:

```
public void visualiza(Client client) {

    validator.checking(new Validations() {{
        that(client.getId() != null, "id", "id.should.be.filled");
    }});
    validator.onErrorUse(page()).of(ClientsController.class).list();

    //continua o metodo
}
```

- UploadedFile.getFile() now returns InputStream.
- EntityManagerCreator and EntityManagerFactoryCreator
- bugfixes

14.5- 3.0.0-beta-4

- New result: result.use(page()).of(MyController.class).myLogic() renders the default view (/WEB-INF/jsp/meu/myLogica.jsp) without execting the logic.
- Mock classes for testing: MockResult e MockValidator, to make easier to unit test your logics. They ignores the fluent interface calls and keep the parameters included under the result and the validation errors.

- The URIs passed to result.use(page()).forward(uri) and result.use(page()).redirect(uri) can't be logic URIs. Use forwards or redirects from result.use(logic()) instead.
- Parameters passed to URI's now accepts pattern-matching:
  - Automatic: if we have the URI /clients/{client.id} and client.id is a Long, the {client.id} parameter will only match numbers, so, the URI /clients/42 matches, but the /clients/random doesn't matches. This works for all numeric types, booleans and enums. VRaptor will restrict the possible values.
  - Manual: in your CustomRoutes you can do: routeFor("/clients/{client.id}").withParameter("client.id").matching .is(ClienteController.class).mostra(null); which means you can restrict values for the parameters you want by regexes at the matching method.
- Converters for joda-times's LocalDate and LocalTime comes by default.
- When Spring is the IoC provider, VRaptor tries to find your application's spring to use as a father container. This search is made by one of the following two ways:
  - WebApplicationContextUtils.getWebApplicationContext(servletContext), when you have Spring's listeners configured.
  - applicationContext.xml inside the classpath

  If it's not enough, you can implements the SpringLocator interface and enable the Spring's ApplicationContext used by your application.
- Utils:
  - SessionCreator and SessionFactoryCreator to create Hibernate's Session and SessionFactory to your registered components.
  - EncodingInterceptor, to change you default encoding.
- several bugfixes and docs improvements.

14.6- 3.0.0-beta-3

- Spring becomes the default IoC provider
- the applicationContext.xml under the classpath is used as Spring initial configuration, if it exists.
- improved docs at http://vraptor.caelum.com.br/documentacao
- small bugfixes and optimizations

Migrating from VRaptor2 to VRaptor3

15.1- web.xml

In order to migrate, in small steps, you'll only need to put on your web.xml:

```
<context-param>
    <param-name>br.com.caelum.vraptor.provider</param-name>
    <param-value>br.com.caelum.vraptor.vraptor2.Provider</param-value>
</context-param>

<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <!-- Your base package here -->
    <param-value>com.companyname.projectname</param-value>
</context-param>
<filter>
    <filter-name>vraptor</filter-name>
    <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
</filter>

<filter-mapping>
    <filter-name>vraptor</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Don't forget to remove the old VRaptorServlet declaration from VRaptor2, and its respective servlet-mapping.

15.2- Migration from @org.vraptor.annotations.Component to @br.com.caelum.vraptor.Resource

VRaptor2's @Component correspondent in VRaptor3 is @Resource. Therefore, in order to make logic classes accessible, just annotate them with @Resource (removing the @Component).

The conventions used are slightly different:

In VRaptor2:

```
@Component
public class ClientsLogic {

    public void form() {

    }

}
```

In VRaptor 3:

```
@Resource
public class ClientsController {

   public void form() {

   }
}
```

The form method will be accessible from the URI: "/clients/form", and the default view will be `WEB-INF/jsp/clients/form.jsp`.

Which means, the suffix `Controller` is removed from the class name and there is no more .logic at the end of the URI. Also, the result jsp doesn't have either "ok" or "invalid" on its name.

15.3- @In

VRaptor3 manages the dependencies for you, so, what you were used to annotate with @In on VRaptor2, you'll only need to receive as constructor arguments:

In VRaptor 2:

```
@Component
public class ClientsLogic {
    @In
    private ClientDao dao;

    public void form() {

    }

}
```

In VRaptor 3:

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    public ClientsController(ClientDao dao) {
        this.dao = dao;
    }

    public void form() {

    }
}
```

In order for this to work, you only need that your ClientDao is annotated with VRaptor3's @br.com.caelum.vraptor.ioc.Component.

15.4- @Out and getters

In VRaptor2 you used either the @Out annotation or a getter method to make and object accessible to the view. In VRaptor3 you only need to return the specified object, if it's only one, or make use of a special object which exposes your objects to the view. This object is the Result.

In VRaptor 2:

```
@Component
public class ClientsLogic {
    private Collection<Client> list;

    public void list() {
        this.list = dao.list();
    }

    public Collection<Client> getClientList() {
        return this.list;
    }

    @Out
    private Client client;

    public void show(Long id) {
        this.client = dao.load(id);
    }

}
```

In VRaptor 3:

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    private final Result result;

    public ClientsController(ClientDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }

    public Collection<Client> list() {
        return dao.list(); // the name will be "clientList"
    }

    public void listaDiferente() {
        result.include("clients", dao.list());
    }

    public Client show(Long id) {
        return dao.load(id); // the name will be "client"
    }
}
```

When your method's return type isn't void, VRaptor uses that type to find out which will be the object's name

on the view. When not using the Result object, the name of the exposed object depends on the method's return type. If the return type is a Collection, the object name will be the name of the object contained by the Collection followed by the word List. In the above example, the object would be named 'clientList'. Otherwise, if the return type is a single object, the exposed object's name will be the name of the class with lowercase characters.

15.5- views.properties

In VRaptor3 there's no views.properties file, although it is supported when using VRaptor3's compatibility mode. Thus, all redirections are made on the underlying logic, using the Result object.

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    private final Result result;

    public ClientsController(ClientDao dao, Result result) {
        this.dao = dao;
        this.result = result;
    }

    public Collection<Client> list() {
        return dao.list();
    }

    public void save(Client client) {
        dao.save(client);

        result.use(Results.logic()).redirectTo(ClientsController.class).list();
    }
}
```

If it's redirection to a logic, you can refer to it directly, and the given parameters will be passed to the called logic.

If you want to forward to a JSP page, you can use:

```
result.use(Results.page()).forward("/WEB-INF/jsp/clients/save.ok.jsp");
```

15.6- Validation

You don't need to create a method called validateLogicName in order to do the validation, you only need to receive the br.com.caelum.vraptor.Validator object in your logic's constructor, and use it to do your validation, specifying which logic to go when your validation fails.

```
@Resource
public class ClientsController {

    private final ClientDao dao;
    private final Result result;
    private final Validator validator;

    public ClientsController(ClientDao dao, Result result, Validator validator) {
```

```
        this.dao = dao;
        this.result = result;
        this.validator = validator;
    }

    public void form() {

    }

    public void save(Client client) {
        if (client.getName() == null) {
            validator.add(new ValidationMessage("error","invalidName"));
        }
        validator.onErrorUse(Results.page()).of(ClientsController.class).form();
        dao.save(client);
    }
}
```

15.7- Putting objects on Session

On VRaptor2 it was enough an @Out(ScopeType.SESSION) for putting an object on HttpSession. It doesn't work on VRaptor3, because this way you lose control on your variables. So in VRaptor3 you have to do one of this two approaches:

- Your object will be accessed only by components and controllers, not by jsps:

  ```
  @Component

  @SessionScoped
  public class SessionMyObject {
      private MyObject myobject;
      //getter and setter
  }
  ```

  And you receive on your classes constructors a SessionMyObject, and use getters and setters to handle the MyObject on session.

- The object will be accessed in jsps too:

  ```
  @Component

  @SessionScoped
  public class SessionMyObject {
      private HttpSession session;
      public SessionMyObject(HttpSession session) {
          this.session = session;
      }
      public void setMyObject(MyObject object) {
          this.session.setAttribute("object", object);
      }
      public MeuObjeto getMeuObjeto() {
          return this.session.getAttribute("objeto");
      }
  }
  ```