Forum: JSF

understanding the basic MVC flow with JSF

john lazeraski
Ranch Hand
Posts: 76

posted 2 years ago

Hi all,

I am learning JSF 2 (going to use JSF 2.2 with GlassFish 4). I really like what I am seeing, but a few things are a little confusing to me. I understand the concept of a JavaBean for a model object. I have done a few examples for a simple login page, which uses a basic username/password string, and even have a template set up to show the non-logged in header (with the login form) and a logged-in header

when the user is logged in.

What I am struggling with at the moment is understanding the basic MVC flow with JSF. I have a command button and h:form, and from examples, it looks like I use EL to set the action to the same login bean that stores my user/password properties, pointing to a method to handle the login. In the past I've been used to separating my model objects from the actual controller (action classes in struts). In this case, it seems that I combine the model of the login object that is stored in the session with the code that handles the login (and logout). Is this the right way to use JSF 2? Or is there a better way to separate the model/properties from the actual code that handles logic, redirection, etc?

Is there any sense in having a separate "model" login object, user object, etc.. and then use separate controller beans (via EL) to handle the logic work and redirections? I just want to stay within the best practices so that if our project grows and we bring on some more JSF developers, my project layout and the way it flows isn't ridiculous.

Thank you.

K. Tsang
Bartender
Posts: 3157

10

I like...

posted 2 years ago

Hello John.

There's Glassfish 4 already? Anyway JSF is any MVC framework as you have guessed. The controller is that Faces Servlet thing you see when you set up web.xml DD

you set up web.xml DD.

JSF uses the concept of "managed bean" (the annotation @ManagedBean) to represent the model of a page. The view is the actual jsf or xhtml page with EL, <h:xxx>, <f:xxx> tags.

Simply put given a form, the action you specify is defined in the managed bean (aka the model) doing CRUD or whatever. The managed bean itself can be the entity POJO (database record) plus extra methods for the view. Or can be separate.

Approach 1 separating managed bean from entity bean

```
1  // database record "USER"
2  public class User {
3    private String userId;
4    private String password;
5    //getters & setters
6  }
7
8  // managed bean
9  @ManagedBean(name="userLogin")
10 @ViewScope // can also be Request or Session
11 public class UserLogin {
12   private User userBean;
13
14   public User getUserBean() { return userBea
15
16   public void setUserBean(User userBean) { t
17
18   public String login() {
19     // do login eg check userId and passwor
20   }
21 }
22
```

```
23  //JSF
24  <h:form>
25  <h:outputText value="Username" /> <h:inputTe
26  <h:outputText value="Password" /> <h:inputSe
27  <h:commandButton action="#{userLogin.login}"
28  </h:/form>
```

Approach 2 using entity bean as managed bean

```
1   // database record "USER"
2   @ManagedBean(name="user")
3   @ViewScope // can also be Request or Session
4   public class User {
5     private String userId;
6     private String password;
7     //getters & setters
8     public String login() {
9       // do login eg check userId and password
10    }
11  }
12
13  // JSF
14  <h:form>
15  <h:outputText value="Username" /> <h:inputTe
16  <h:outputText value="Password" /> <h:inputSe
17  <h:commandButton action="#{user.login}" />
18  </h:/form>
```

As you can see the difference is in the value attribute "userLogin.userBean.userId" vs "user.userId" in the h:inputText tag and the action in the h:commandButton.

I personally prefer separate the managed bean from the entity bean especially when the form has many inputs which can clutter the code.

Hope this helps.

K. Tsang JavaRanch SCJP5 SCJD OCPJP7 OCPWCD5 OCPBCD5 OCPWSD5 OCMJEA5 part 1 part 2/3

john lazeraski
Ranch Hand
Posts: 76

posted 2 years ago

Hi,

Thank you for the reply.. exactly what I was looking for. I too like the separation. What confused me is all MVC frameworks have a single controller servlet that then pass on to some class somewhere the request. I have been reading a few books and examples on JSF and thus far nothing really stuck out to indicate that you use the bean as both a pojo AND the logic... or I should say it came across to me like that but having done Struts, Spring and also using REST for ajax calls, it wasn't abundantly clear that this was the way it works in JSF. I was looking for that "action" class. So thank you, this makes sense, and as long as JSF 2.2 follows the same semantic, that's great.

Yes, GlassFish 4 is now out, as of last week, and JEE7 is fully implemented including JSF 2.2. I like what JSF 2.2 brings with the new flow stuff, but no books on it yet so I'll need to use whatever is on the web to figure out how to utilize it.

Tim Holloway
Saloon Keeper
Posts: 17487

30

I like...

posted 2 years ago

OK. I didn't read all that as closely as I should have (too early in the morning), but I think I can give some general guidelines. But before I start: User-designed login/security systems are **dangerous**. Unless you are a full-time security professional, odds are virtually 100% that there will be holes in what you design, and if you're like about 90% of the people I've seen do this over the last 10 years or so, your system will have a flaw that non-technical people can break through in about 10 minutes.

The J2EE standard defines a security subsystem. It's standard equipment on every J2EE server, even the most minimal ones. It was designed by full-time security experts, is fully debugged, and as far as I know, no one has ever broken through it. And, it's integrated into the J2EE API.

OK. Now on to MVC. JSF is just about the purest form of MVC you'll find on a web platform. The only difference between it and an ideologically-pure MVC implementation is that which is imposed on it by the HTTP protocols, which is that changes to the Model cannot be pushed out to the View, since HTTP can only handle requests, not unsolicited responses. There is a name for this less-interactive MVC paradigm, but I cannot remember it.

One of the things about MVC is that there doesn't have to be a single View, a single Controller, or even a single Model. You can, for example, have both a spreadsheet and a pie chart referencing the same Model and updating the spreadsheet would reflect to the Model, which would

reflect to the pie chart (or vice versa). You can also have a hierarchy of MVCs so that complex View controls could be contstructed from simpler sub-Views. A classic example of this is the ComboBox control, which is an amalgamation of a straight textbox and a SELECT dropdown list. The sub-views are paired with sub-controllers.

That's just general MVC theory at work. In JSF, the master Controller is always the FacesServlet. Sub-Controllers are incorporated into the various control element tag implementations. You almost never write controller code in JSF, because it's all pre-supplied. So you only have to supply the View templates (xhtml) and the Models (backing beans).

A lot of people think that the action logic in backing beans makes them Controllers. This is incorrect. A Controller is a component whose sole purpose in life is to synchronize the Model and View. In JSF, that task is performed by the FacesServlet and the controls. You may have Validators and Converters performing adjunct functions, but the actual synchronization (updating) is part of the JSF core.

So what are action methods? Well, actually MVC is fairly useless by itself. All it does is sit there all day long and keep the Model and View in sync. In the real world, an MVC needs to be attached to some sort of business functionality that gets data from the Model in and out of external locations (such as databases). Or performs calculations. Or navigates to other Views. Or all of the above and then some. In other words, the Action methods are business-support methods external to the MVC functionality of JSF.

So a JSF backing bean is typically a polluted Model, if you like. Purists might object to this, but the purpose of the Model is to support the GUI, and by having the action methods within the model, they have direct access to the model properties, reducing the number of code modules required. Unlike ORM models, the JSF Models are not expected to be completely JSF-independent (although the more they are, the better). So the fact that they incorporate some business/navigation functionality is no major issue. Although good design says that anything really complex along those lines should reside in external bean(s) dedicated to that purpose.

And, finally. JSF is based on a state machine which tracks a lifecycle from the receipt of an incoming HTTP request to the transmission of the corresponding response. It is essential for anyone who wants to become competent in JSF to know that lifecycle and to understand the primary and secondary paths through the JSF lifecycle.

One thing more. Some rules to live by:

1. JSF is designed to wrap functionality around POJOs. The more JSF-specific code you have written (excepting JSF model objects) in a backing bean, the more likely you did it wrong.
2. EL is not intended as a programming language. Its primary use with JSF is to define references to JSF backing bean properties and actions. If you code application logic on the View, you not only pollute the idiotlogical purity of the MVC paradigm, you create a  mess, where you never know if logic is located in the Model or the View or in some unholy interacting cross-connection of both. Complex EL

statements are also a royal pain to debug.
3. No, you shouldn't mix JSP's JSTL and JSF view definitions. Even though Oracle's documentation may suggest it. JSTL doesn't play that well with JSF, and almost anything that you can do with JSTL has a much cleaner JSF equivalent.

An IDE is no substitute for an Intelligent Developer.

john lazeraski
Ranch Hand
Posts: 76

posted 2 years ago

Wow! Thank you for such a great informative reply.

I am well versed in MVC but as you pointed out, my understanding was much more in tune with Struts and others where I write the "extension" controller code.

I am glad my understanding of how to use EL is in line with what you said. I do not want to pollute my classes with the old JSP style scriplets and such. I was not totally aware of JSTL at this point but with your reply and Mr Tsang's I have a much clearer picture of MVC with JSF, although I am not quite sure yet how I do looping and conditionals without it? I don't know enough of the JSF components and such to understand how I won't need JSTL for loops and conditionals.

I am not sure if you're familiar with Ext-js? I was originally going to use it, but was told it's not nearly as performance capable for potentially

well trafficked sites. I liked that it had really nice ajax and UI controls like pop up dialogs and a slick layout system that took care of cross platform and browser layout issues.. of which my expertise lies on the back end and not in the front UI. Is there anything within the JSF framework that mimics what Ext-JS offers in the way of pop up dialogs, layouts and such? I assume not as I haven't seen anything with regards to that.. I believe I need to handle all the CSS, layouts of components and such myself?

As for form actions, I am doing as Mr TSang mentioned.. I have a pojo that is JUST for model data, and a separate backing bean with the login method that uses a private instance of my pojo. In my form I have like he showed, #{login.user.username} for the input value. Does that sound right with what you are saying.. separating the pojos from the beans as much as possible? Also, perhaps I am doing things wrong, but in my previous use of Jersey with REST representations, I shared the model pojo with the rest representation class AND the back end entity bean using model delegation. For example I had something like this:

```
1  public class User {
2    private String name;
3
4    public String getName(){
5      return name;
6    }
7
8    public void setName(String name){
9      this.name = name;
10   }
```

```
11  }
12
13  @Entity
14  @Table(name="users")
15  public class UserEntity {
16    private User model;
17
18    public UserEntity(){
19      this.model = new User();
20    }
21
22    public UserEntity(User model){
23      this.model = model;
24    }
25
26    @Transient
27    public User fetchModel(){
28      return model;
29    }
30
31    @Id
32    @Column(name="name")
33    public String getName(){
34      return model.getName();
35    }
36
37    public void setName(String name){
38      model.setName(name);
39    }
40  }
41
42  @XmlRootElement(name = "user")
43  public class UserRepresentation {
44    private User model;
45
46    public UserRepresentation(){
47      this.model = new User();
48    }
49
```

```
50    public UserRepresentation(User model){
51      this.model = model;
52    }
53
54    public User fetchModel(){
55      return model;
56    }
57
58    public String getName(){
59      return model.getName();
60    }
61
62    public void setName(String name){
63      model.setName(name);
64    }
65  }
```

The above allowed me to share a common pojo model with the front facing REST representation class (which we hand coded rather than use JAXB to generate from an XSD for other reasons as well) AND the back end entity bean. When ever we modified the model, we'd just modify the representation and entity beans as well. It would allow me to handle a rest call to say create a user like so:

```
1   @Path("users")
2   @Stateless
3   public class UserResource  {
4     @EJB
5     UserBean userBean;
6
7     @POST
8     @Consumes({MediaType.APPLICATION_JSON, Mec
9     public Response create(UserRepresentation
```

```
10          userBean.create(user.fetchModel());
11        }
12      }
13
14      @Stateless
15      public class UserBean {
16          @PersistenceContext(name = "somePU")
17          EntityManager em;
18
19          public User createUser(User user) {
20              UserEntity ue = new UserEntity(user)
21
22              em.persist(ue);
23              em.flush();
24              em.refresh(ue);
25              return ue.fetchModel();
26          }
27      }
```

OK..so the reason I wanted to share this is to ask you if this seems to make sense in that I am sharing a POJO (separting it not only from the JSF annotations, but also the entity annotations) with the JSF (or in my example, REST representation) and the back end database. It also would allow me to more rapidly switch out either the front end or back end without breaking the model. I like the way this works out, and the only caveat is the little bit of extra code in each method to use delegation to the underlying model.

I think this fits well with what you are saying too.. in that I am breaking out the underlying model pojo from the rest of the system, keeping it clean and not dependent on anything specific. What do you think?

I do have a question.. something I am about to try out.. reading through the JSF 2.2 and JEE7, it looks like a big part of JEE7 is allowing for CDI in more places in the code. I know in JEE6, I could use @EJB (and @Resource) in servlets, and thus anything extending them including my resource classes like the example I have above. So can I use @EJB and @Resource (and I guess @Inject) inside of JSF backing beans? I want my login() method in the JSF backing bean to be able to call the EJB session bean that uses the user entity bean to look up the username and password. Is that now allowed (or was it even in JSF 2.x and JEE6)? If not, how then would a login method in the backing bean be able to call the EJB session bean to work with entities? I'll be trying this soon myself and may discover my own answer, but just in case I am hoping it can be answered here.

Also, I totally agree with you in using the JEE security model.. however I've never done that and I am not sure where best to discover how to implement that in a pretty JSF UI while using the JEE container login process? I've added a ProtectedResourceFilter to the web.xml filter section, so ensure nothing in my /secured/* can be reached IF the user is not logged in (which will result in a Session object in the HttpSession if they are). This is something I found online and looked reasonable to protect "inside" resources. Is this OK..or is this ALSO a bad way to do it and the JEE/Realm stuff should be used here as well? If you have a link or two that could point me to info on how best to implement that (especially in JEE7 if it's improved over JEE6), that would be fantastic.

Something else that EXT-JS had that I thought was nice is the notion of fragments.. or maybe I am forgetting the word now.. but the idea was you could put together a few pages, backing code, etc.. and then insert that on any page where you wanted and it worked alone (you would pass in data no doubt). I am not sure if it could interact with other elements on the page, but it made it very reusable whenever you needed it. Naturally this would be like any JSF component now, but you could build up your own custom UI and code, and insert it as a fragment. I am currently using a template for my basic layout (header/content/footer) but thinking templates are a bit too abstract for what I am referring to. I'd want to not have to change a template layout to insert a fragment in some pages but not others. I'd even like the fragment to be potentially dynamically inserted.. perhaps a user selects something or changes something and now the fragment can be shown. I don't know well enough if templates are that dynamic but I think from what I've read you have to define them ahead of time. Giving a user say a selection of cool components they could just put wherever would make a template not the ideal choice in this case. So is there anything like a fragment where I can build up say a number of components into one fragment and reuse on any page?

> “
> And, finally. JSF is based on a state machine which tracks a lifecycle from the receipt of an incoming HTTP request to the transmission of the corresponding response. It is essential for anyone who wants to become competent in JSF to know that lifecycle and to understand the primary and secondary paths through the JSF lifecycle.

Yes.. I understand it to some degree, the request phase, validation phase, render phase, etc. But I am not an expert with it yet. This sort of reminds me of Android life cycle. I'll definitely learn it as I work with JSF.

One thing more. Some rules to live by:

> “
> 1. JSF is designed to wrap functionality around POJOs. The more JSF-specific code you have written (excepting JSF model objects) in a backing bean, the more likely you did it wrong.

I assume this fits well with my example above if I understand you correctly? I completely put NO anything in my model pojos, just pure pojos, then with delegation and in the case of my login bean with the login method, it references the User pojo. Thus, did I do it correctly?

> “
> 2. EL is not intended as a programming language. Its primary use with JSF is to define references to JSF backing bean properties and actions. If you code application logic on the View, you not only pollute the idiotlogical purity of the MVC paradigm, you create a  mess, where you never know if logic is located in the Model or the View or in some unholy interacting cross-connection of both. Complex EL statements are

also a royal pain to debug.

3. No, you shouldn't mix JSP's JSTL and JSF view definitions. Even though Oracle's documentation may suggest it. JSTL doesn't play that well with JSF, and almost anything that you can do with JSTL has a much cleaner JSF equivalent.

As I asked above, if I use pure EL with no JSTL, I am sure I'll learn but I assume there is a replacement for loops and conditionals? I think I saw the render attribute on components which when bounded with a backing bean property could conditionally control if it's displayed or not? I assume also then that putting a number of components (or like my question about a fragment above) inside a panel and tying it all with the panel redner attribute to a property in a bean, an entire section of components could be displayed/hidden?

The last remaining thought of mine is how to do this all with ajax? For example, instead of waiting to render or not redner a component with a bounding property, how about hiding/showing elements based on ajax requests/repsonses? That is.. I have a drop down with YES and NO. The user selects YES and a fragment shows up without a page refresh/redirect... they select NO and it disappears? I know how to write some javascript code to hide/show ids/classes.. I am just wondering if the JSF AJAX support can handle this for me without having to write javascript code myself? If so, I don't include any javascript code myself do I? Or do I need to say, include a jquery.js

file to get some of this to work?

Thank you again for so much good info and appreciate any responses you can provide to my questions above.

Tim Holloway
Saloon Keeper
Posts: 17487

30

I like...

posted 2 years ago

I'm sure I'm going to miss a lot here. Once a message gets larger than my screen, I lose track of things.

But here's a few. First, if you use J2EE container login, the login screen should **not** be a fancy JSF page. For starters, since the login is processed by your webserver and not by application code, not all containers will route the login through the FacesServlet, without which a lot of JSF functionality cannot work. For another, the more extraneous stuff that's on a login page, the more things there are that can potentially be exploited. So my login pages have all the warmth of a TSA checkpoint manned by tax auditors. Which is to say a stark HTML or JSP form for the entry of user ID and password and that's about it. Any attempt to appear friendly can be used by an exploiter to claim that their intentions were welcomed, and I don't want to do that.

JSF backing beans are POJOs. That was a central design point. They'd seen how Struts had suffered from a requirement to base models on specialized classes and interfaces and decided that there had to be a better way. Doing the model as a 2-layer construct is fine when you want to make a third-party POJO like from an ORM part of the mix, since JSF's instantiation methods aren't conducive to using EJB Entity

beans directly as backing beans. But there's no benefit to breaking a perfectly good POJO into 2 parts for no reason.

I'm familiar with the name Ext-JS, but I forget the details. Many of the third-party JSF extension tagsets such as RichFaces and PrimeFaces pull in jQuery to help them do their work (especially the AJAX stuff). Most javascript frameworks will play well with JSF, although extensions such as the ones I just mentioned also have "smart controls", so overall you don't need as much explicit client-side logic. RichFaces and PrimeFaces both include pop-up dialog support. I don't have an exact list of any of the other frameworks' capabilities, but I won't rule any of them out.

"Looping" and "conditional" constructs on the View are trouble. JSF Views are defined in terms of geometry, not logic. When you need a tabular component, you don't "loop through" data to render the table, you use a JSF 2-dimensional control. Whether that control functions by looping, parallel processing all the rows at once, or something even more arcane doesn't matter, since the goal isn't logic, it's a 2-dimensional display element.

"Conditional" should be limited to display elements only, where the JSF "rendered=" attribute can show or hide components and their children (if any). Business-related conditional logic has no place in the View. That goes in the backing bean. Most dynamic display functions can be done via display-element properties such as "rendered=" or by changing the backing model data. Actual dynamic construction of GUI elements "on the fly" is nowhere near as needful as too many people

think it is. I've only got 2 webapps that do that, and one of them probably shouldn't.

And yes, JSF - especially JSF2 - is designed with the idea that you can build up complex re-usable components. You can do this by designing your own custom tags or by creating xhtml resources. Of the 2, the xhtml version is what I recommend for most cases. It can be a real pain to implement a custom tag in Java code, and you'll be accessing resources whose future form isn't yet perfected.

As for rules to live by:

1. No, you don't need to create a complex network of components for a Model. A single POJO-style bean holding the UI model data and action methods is sufficient in most cases. When I said that JSF models should avoid non-model JSF constructs, I didn't mean move them somewhere else, I meant don't use them anywhere. Unless you absolutely must. I'll be glad to cover some details on that later.

2. I've already mentioned that you should only code UI-specific logic on the View (and as little of that as possible). However, to repeat myself, the third-party extension tagsets are very useful to minimize or even eliminate the need for JavaScript. Also, in JSF2, there's a core tag for AJAX, although the older frameworks such as RichFaces 3 gave much the same functionality only using proprietary extensions.

An IDE is no substitute for an Intelligent Developer.

john lazeraski
Ranch Hand
Posts: 76

posted 2 years ago

Thank you. Excellent info. If you find time later, can you provide a modified User snippet that I provided to explain how you would combine the pojo + login functionality WITH ejb? I suspect what you mean is using EJB entity beans + methods as the backing bean for JSF.. in other words, my UserEntity, looks as it does, but would not delegate..just store the properties, and would also have the method login(), and also an @Named("login") in and the scope (in my case probably @ViewScoped using CDI package)? Something like:

```
1  @Entity
2  @Table(name="users")
3  @Named("login")
4  @ViewScoped
5  public class User {
6      .. properties and get/set methods
7
8      public void login(){
9        .. code to try to log in
10     }
11 }
```

Leaving out a bit for brevity, but is that the idea, making the entity also the backing bean and sharing all the properties? Of course, if I was re-using complex components inside of one form, that was based off of different entity/backing beans, I suppose that would not fit this very well, not sure yet how that would work out.

Thank you again.

john lazeraski
Ranch Hand
Posts: 76

posted 2 years ago

Hi again,

I was re-reading your replies on the login/auth stuff. I am a little confused however. I have been looking up links to JEE6/7 realm/security, and one I came across seemed to indicate that you could do form authentication with a JDBC Realm (http://stackoverflow.com/questions/2206911/best-way-for-user-authentication-on-javaee-6-using-jsf-2-0). I would assume then from their example that I could style a form however I want using JSF2 and with a form submit for login, it would utilize the JEE container security? Does that link basically do what you are saying? Or were you saying that I should be using the old style pop-up browser dialog for username/password for protected resources and such? I much prefer the idea that if for example someone logs in, and bookmarks a page, then later comes back to that bookmark, that it automatically takes them to the outside page (the main page in our site has a username/password input box in upper right corner) so that they have to log in again. As well I want a user to see the user/pw fields at all times on the outside site and can log in at any point. Is that possible with the JEE security (or for that matter does that link above basically do that)?

Thanks

Tim Holloway
Saloon Keeper
Posts: 17487

I like...

posted 2 years ago

J2EE authentication is **Container-Managed** authentication (login).

The Realms are plug-compatible. The webapp itself neither knows nor cares whether the Realm is JDBC, LDAP, or Western Union to dwarves living in caverns under the Black Forest. Except for a few settings in the WEB-INF/web.xml file, the webapp won't even know if you are using form-based or basic (popup dialog) authentication.

The actual login code is part of the webapp server. You do not write login code. The server's login code is designed to automatically activate when a user requests an authenticated URL but the user has not yet authenticated (logged in). When this condition is detected by the server, the original URL request is placed on hold, the server looks in the webapp's web.xml file for the location of a login form, and presents that form to the client. When the client submits the form back to the server, it intercepts the form (so that it does not go to the webapp), extracts the j_username and j_password fields from it, and uses them as parameters to the Realm's authenticate() method, which returns a yes/no answer. If the answer is "no" (authentication failed), the server looks in web.xml for the loginfail page and presents it, processing it just like it did the login page. My login and loginfail pages are almost identical, except that I add the message "Login Failed" to the loginfail page template. Once the user is authenticated, the Realm constructs a UserPrincipal object, creates an HttpSession if one does not exist, and binds the UserPrincipal to the HttpSession so that the current and future requests can be populated with the remoteUser and userPrincipal properties, which are then accessible to application

code as needed. Although the internals of UserPrincipal are mostly "black box", so about the only useful thing most application code can do with a UserPrincipal object is query it for its remoteUser property value. At this point, the **original** URL request is taken out of where it had been shunted aside and submitted to the webapp for processing.

There are a couple of things worth nothing here.

1. Using this scheme URLs are bookmarkable, since no matter where in the webapp the URL is referencing, the container (server) is the ultimate gatekeeper. Any attempt to access the URL via a bookmark would trigger the authentication process.

2. The login process is transparent. The user is NOT routed to a "home page" after, login. Instead, processing proceeds on to the originally-requested URL as though login had never occurred. This makes them truly bookmarkable.

3. There are no listeners or other mechanisms that can be invoked to let the webapp know that the user has logged just in. In fact, if you are using a Single-Signon (SSO) Realm, the actual login may have been done on some other server at some other time. For example, for a Windows user, an SSO realm can be set up when the user logs into Windows itself. All I'm really saying is #2 all over again: login is completely transparent to the webapp.

In actuality, there are ways to hack around #2 and #3, but I recommend thinking carefully before doing so. Especially since if I

bookmark a page, there's a reason why I bookmarked that page and not the Home Page, and I tend to get annoyed if I'm re-routed.

An IDE is no substitute for an Intelligent Developer.

john lazeraski
Ranch Hand
Posts: 76

posted 2 years ago

Tim,

Thank you again. I believe I understand what you are saying. My only concern is that I do NOT want a browser like pop-up login window to appear. I want the UI to show a login button OR as I have it now, in the upper right corner I have the username/password fields there and a user can just log in directly from any of the outside pages. I found that link above which seems to indicate that I can provide my own form and setting the realm in Glassfish to JDBCRealm and in the web.xml for form authentication, that I can provide my own fancy looking login panel (or in the example, simple username/password). That is my primary concern. This is a public facing web site. If it were an internal or admin based site, I would be fine with any ugly pop-up login window. But because it's like many sites with login access requirements, I need the login form to blend in with the UI. Is that possible with what you are saying?

Thank you.

Tim Holloway
Saloon Keeper
Posts: 17487

30 

I like...


posted 2 years ago

The difference between BASIC authentication and FORM-based authentication is that in BASIC authentication, the appserver sends back a special HTTP response that tells the user's browser to pop up a login dialog and send back the userid/password from the dialog. How ugly the dialog is depends on how ugly their browser is, but the dialog can only return userid and password back to the server because that's what's in the HTML spec. The user's browser handles the login display/capture as part of its inherent function.

FORM-based authentication is likewise based on the constraint that the only user-submittable data is the user id and password. You provide the template for the login (and loginfail) screen, but the actual form logical components and their processing are strictly defined by the J2EE standard. You cannot add arbitrary fields or functions nor omit (or mis-name) the required components. You also cannot specify a URL to target the user towards this page, since the page is not managed by the web application and therefore doesn't have a true URL. If someone supplies a URL path that can be resolved by fetching the form resource, the form will display, but it won't function. The only way to get a true login page is to submit a URL for a protected resource.

Beyond that, you have free rein to make your login page as pretty or as hideous as you like. For best results, it should be an HTML or vanilla JSP page, not a JSF View, but beyond that, it's whatever you want as far as the purely decorative aspects go.

Just remember that when the login form is presented, the user is not yet logged in, so don't expect to be able to display any resources that aren't publicly available (to non-logged users).

An IDE is no substitute for an Intelligent Developer.

I agree. Here's the link: http://aspose.com/file-tools

Post Reply

New Topic

Similar Threads

- Difference between servlet and struts
- Struts Training
- ask for login
- Confused - MVC, JSP, Servlet interaction
- about struts