Authentication parameters

Broadly speaking, Lift has three LocParams that are used for access control: Test, TestAccess, and HttpAuthProtected. (If can also be used quite successfully.) These names are a little confusing so let’s take a moment to outline their functionality.

**Test and TestAccess**

There are two ways of thinking about results from access control systems in web applications. One would be that, if the user does not have permissions for that part of the application, it should not even appear to exist for them; anything they don’t have access for should simply resolve as HTTP 404 Not Found. This is the route that Test takes. The other line of thought would be that it’s OK to admit the application is there, rather, direct that user to a login or some other friendly approach that won’t baffle the user; this is the route TestAccess adopts. Depending on your application requirements, you have the choice in Lift to serve either implementation choice.

Test differs from TestAccess in that it takes a Req instance as its function argument so you can check any aspect of the request that you like. For example, perhaps the incoming request must have a particular cookie or can only be accessed under a certain host (like localhost). All of these things are easily done with the Test parameter. Consider the following usage:

```scala
>> Test(req => req.hostName == "localhost")
```

In this example, unless the URL is being accessed under http://localhost/, Lift will yield a 404. You can try this out simply by visiting http://127.0.0.1/ and then http://localhost/. The former will get a 404, whereas the later will give you the page as expected. Pretty neat.

Test has its uses in certain applications domains but, on the whole, you are likely to use TestAccess more because it removes items from the menu that yield a Full[LiftResponse] from the function. TestAccess allows you to check within your function any resource you’d like in order to verify that the current user has access to that aspect of the application. For example, you may need to check if a user is logged in and if their particular value exists in the current session. If the user then happens to navigate to that page (perhaps it was saved as a bookmark), you can then safely redirect them to the login URL. Listing 1 demonstrates forcing a user to log in to a page.
For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/perrett/](http://www.manning.com/perrett/)

### Listing 1 Using TestAccess to force user login

```scala
object LoggedIn extends SessionVar[Box[Long]](Empty)

>> TestAccess(() => LoggedIn.is.choice(
  x => Empty)(Full(RedirectResponse("login"))))
```

1. SessionVar to hold check result
2. Yield Empty if session value set
3. Yield LiftResponse if no value

There are several things going on here. For the sake of example, we assign an object called `LoggedIn` that holds a fictional user ID assigned by a page called "login". The `TestAccess` parameter then checks for the existence of the properly set `SessionVar` before allowing access to the page. If `TestAccess` is Empty, the menu is displayed and all is well. If it yields the redirect response, the link is removed from the rendered menu and any attempt to access the page will result in a redirect to the login. There, they may enter their credentials, the code would set the value of `LoggedIn`, and they would be allowed to access the page.

### HttpAuthProtected

In order to use the HTTP authentication within Lift, you must first define something like listing 2 in your Boot class in order to define the type of HTTP authentication you wish to use. You can choose between HTTP Basic Auth, and the more sophisticated HTTP Digest Auth. This is not an issue because it would likely be problematic to mix the two authentication formats.

### Listing 2 Configuring HTTP authentication in Boot

```scala
LiftRules.authentication = HttpBasicAuthentication("yourRealm"){    
  case (un, pwd, req) => if(un == "admin" && pwd == "password"){    
    userRoles(AuthRole("admin")); true    
  } else false
}
```

1. Define your realm
2. Check credentials
3. Allow or deny access

In this sample, I have chosen to show HTTP Basic Auth, but the implementation would be nearly identical for HTTP Digest Auth (#1). One would hope that your credential checking would be significantly better than this but, essentially, the point is that you just need to check what the user has actually entered (#2). Whether you look that up in a database, or some configuration file within your app, or indeed any other way you can think of is really not the concern of Lift. Ultimately, you have to return true to allow the user access to this page (#3) or false to deny them.

With that in mind, the implementation of the `HttpAuthProtected` parameter is very simple. Consider the following:

```scala
>> HttpAuthProtected(req => Full(AuthRole("admin")))
```

You might be wondering about this role malarkey that was also used within listing 8.3. Essentially this structure allows you to define a simplistic role structure. `userRoles` is just a `RequestVar` with a `List[Role]`. In this instance, I have assigned everyone to be the "admin" role, and then in the `LocParam` usage the `Full(AuthRole("admin"))` tells Lift that users must qualify as Admin in order to gain access to this page.

### External security models

Before finishing this article I just wanted to touch on external security models. Many people coming to Lift from an enterprise Java background often wonder why Lift does not ship with integrations into products such as Spring Security. Well, the reason for this is mostly an ideological one. The model that is provided within Lift essentially allows you the following security construct:
Consider for a moment that such a function can be applied to pages, locations, URLs, and even field level access within Mapper. The Lift ethos is that this construct can be applied throughout your application to whatever degree of granularity you require. With integrated access control, you always know what the access rules are for a given resource.

Lift is evolving constantly and modules for things such as OpenID are continually being added and improved upon. Technically speaking, there are no issues in using an external security model; rather, with SiteMap protecting your pages, you will never present the user a link that is not accessible for them, nor will you have to worry about parallel logic between your menu rendering and your external access control system as would be the case with an off-the-shelf-solution.

**Summary**

Lift provides you with a comprehensive set of tools that you can use to determine how and when a user should be allowed access to a page. If you require more, you can always either write your own location parameter or just extend sitemap locations.
Here are some other Manning titles you might be interested in:

- **Scala in Action**
  Nilanjan Raychaudhuri

- **AspectJ in Action, Second Edition**
  *Enterprise AOP with Spring Applications*
  Ramnivas Laddad

- **DSLs in Action**
  Debasish Ghosh

Last updated: March 1, 2011