Spring Login :

Imp URL:

<http://docs.spring.io/spring-session/docs/current/reference/html5/>

<https://docs.spring.io/spring-security/site/docs/3.0.x/reference/session-mgmt.html>

<http://forum.spring.io/forum/spring-projects/security/80712-custom-session-management-filter-logging-out-throws-redirect-exception>

<http://codehustler.org/blog/spring-security-tutorial-form-login/>

----------------------------=================++++++++++++++++++++++\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Adding a custom object into the session**

In this section I’m going to show you how to add a custom object into the session, and how to display еру object’s properties on JSP. The role of the custom object will play the *Person* object. Firstly I’ll modify the existing person controller:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | @Controller  @SessionAttributes("personObj")  public class PersonController {        @RequestMapping(value="/person-form")      public ModelAndView personPage() {          return new ModelAndView("person-page", "person-entity", new Person());      }        @RequestMapping(value="/process-person")      public ModelAndView processPerson(@ModelAttribute Person person) {          ModelAndView modelAndView = new ModelAndView();          modelAndView.setViewName("person-result-page");            modelAndView.addObject("pers", person);          modelAndView.addObject("personObj", person);            return modelAndView;      }    } |

Comparing with the latest version I have added two new strings:

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  @SessionAttributes("personObj")  ...  modelAndView.addObject("personObj", person); |

--------===========-----------------------)))))))))))(((((((((((%%%%%%%%%%%%%%%%%@@@@@@@@

What you want is to add a custom authentication provider to spring's normal authentication manager. So you would configure the authentication manager like this:

<security:authentication-manager alias="authenticationManager">

<security:authentication-provider user-service-ref="authServiceImpl">

<security:password-encoder ref="passwordEncoder"/>

</security:authentication-provider>

</security:authentication-manager>

<bean id="passwordEncoder" class="org.springframework.security.authentication.encoding.Md5PasswordEncoder"/>

Now you only need to define the authServiceImpl bean inside your spring context. You can either do this through xml or annotations (my prefered way).

@Service

public class AuthServiceImpl implements AuthService {

You need to implement the AuthService interface. Just implement to methods from the interface - should be pretty straight forward. You don't need to put things into the SecurityContextHolder yourself - spring will do that.

What you want is this:

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException, DataAccessException {

return MyUser user = myUserService.getMyUser(username);

}

Feel free to ask if you have any further questions.

EDIT: Or you could just have your UserService class implement the interface - I just did it like this because you didn't provide your UserService class.

----------------\*\*\*\*\*\*\*\*\*\*\*\*\*%%%%%%%%%%%%%%%%%%%$$$$$$$$$$$$$$$$$$$$$

# Using Http Session With Spring

There are multiple ways to get hold of and use an Http session with a Spring based web application. This is a summarization based on an experience with a recent project.  
  
**Approach 1**  
  
Just inject in HttpSession where it is required.

@Service

public class ShoppingCartService {

@Autowired

private HttpSession httpSession;

...

}

Though surprising, since the service above is a singleton, this works well. Spring intelligently injects in a proxy to the actual HttpSession and this proxy knows how to internally delegate to the right session for the request.  
  
The catch with handling session this way though is that the object being retrieved and saved back in the session will have to be managed by the user:

public void removeFromCart(long productId) {

ShoppingCart shoppingCart = getShoppingCartInSession();

shoppingCart.removeItemFromCart(productId);

updateCartInSession(shoppingCart);

}

**Approach 2**  
  
Accept it as a parameter, this will work only in the web tier though:

@Controller

public class ShoppingCartController {

@RequestMapping("/addToCart")

public String addToCart(long productId, HttpSession httpSession) {

//do something with the httpSession

}

}

**Approach 3**  
Create a bean and scope it to the session this way:

@Component

@Scope(proxyMode=ScopedProxyMode.TARGET\_CLASS, value="session")

public class ShoppingCart implements Serializable{

...

}

Spring creates a proxy for a session scoped bean and makes the proxy available to services which inject in this bean. An advantage of using this approach is that any state changes on this bean are handled by Spring, it would take care of retrieving this bean from the session and propagating any changes to the bean back to the session. Further if the bean were to have any Spring lifecycle methods(say @PostConstruct or @PreDestroy annotated methods), they would get called appropriately.  
  
**Approach 4**  
Annotating Spring MVC model attributes with @SessionAttribute annotation:

@SessionAttributes("shoppingCart")

public class OrderFlowController {

public String step1(@ModelAttribute("shoppingCart") ShoppingCart shoppingCart) {

}

public String step2(@ModelAttribute("shoppingCart") ShoppingCart shoppingCart) {

}

public String step3(@ModelAttribute("shoppingCart") ShoppingCart shoppingCart, SessionStatus status) {

status.setComplete();

}

}

The use case for using SessionAttributes annotation is very specific, to hold state during a flow like above  
  
  
Given these approaches, I personally prefer Approach 3 of using session scoped beans, this way depending on Spring to manage the underlying details of retrieving and storing the object into session. Other approaches have value though based on the scenario that you may be faced with, ranging from requiring more control over raw Http Sessions to needing to handle temporary state like in Approach 4 above.

*@SessionAttributes* doesn't fully replaces the traditional *HttpServlet* session management. Use it if two or more Controller methods need to communicate some data. But, using this we can only achieve communication within single controller class. You do not use to read and write from and to the session explicitly if you are using *@SessionAttributes*. Usage of *@SessionAttributes* is suggested only for short lived communications. If you need to store long term data in session, it is suggested to use*session.setAttribute* and *session.getAttribute* explicitly, instead of *@SessionAttributes*. For more information