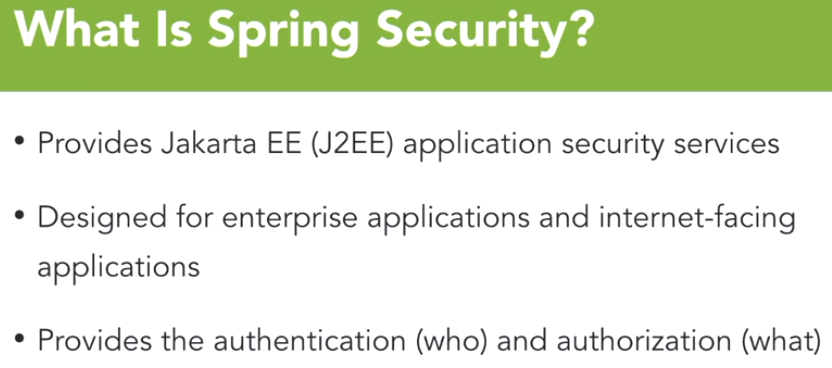


Security should be tackled in a layered fashion. Layered model starts at the lowest level and that is the physical hardware. Hardware must be physically secured and uniquely identifiable to other trusted systems.

After you plan to secure the hardware, you start looking at the network. Here we deploy transport-layer security, firewalls, network segmentation strategies like VLANs and security zones, IDS systems

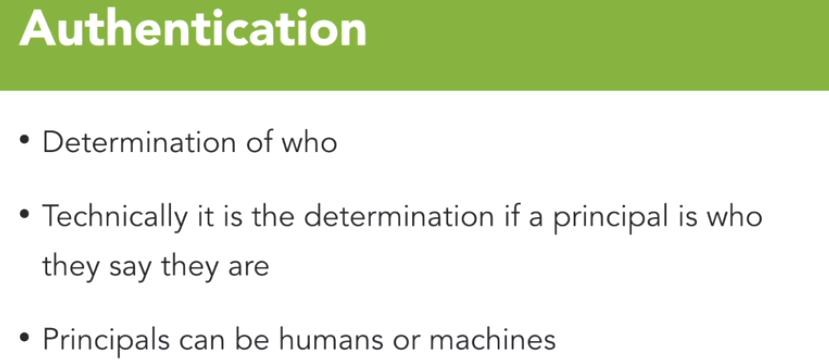
 We then move to the operating systems, where user access controls, patching policies, and software restrictions provide additional layers of security for our system. Finally, we look at application security.

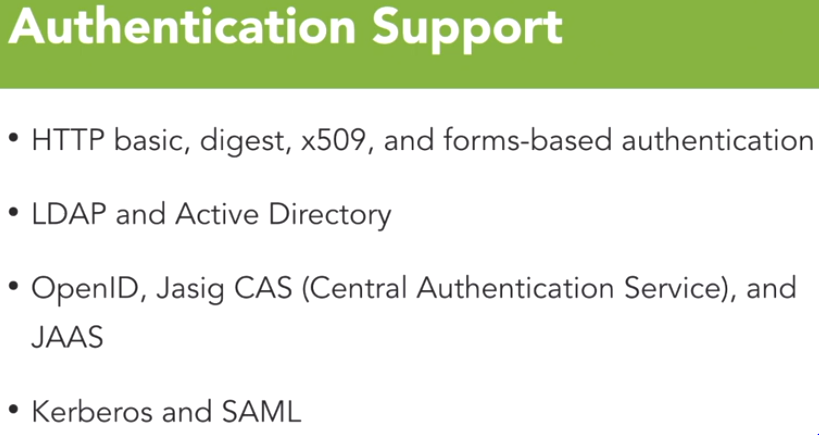
And this is where we focus on good coding practices, proper data handling, and application user access controls. And this is where Spring Security comes into play.



Spring Security is a project that provides amazing abstractions in the J2EE, now called Jakarta EE, application space. These abstractions help solve the various issues of traditional Java enterprise security like portability and vendor lock-in.

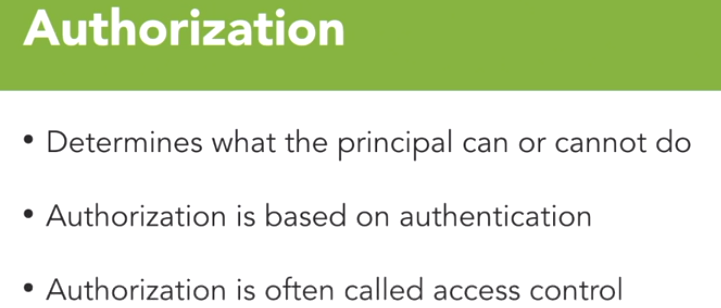
Spring Security isn't just for enterprise developers. It also provides valuable tools for internet facing applications running in the JVM.



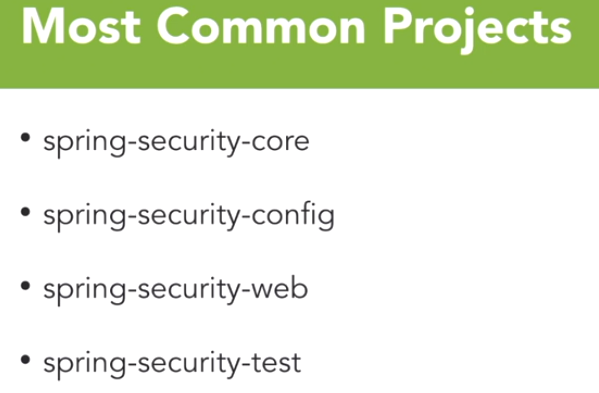
Spring provides out of the box support for many different flavors of authentication. In the Web world, there's support for http basic, as well as http digest authentication. It also includes support for x509 certificate-based authentication. The most common, internal level of authentication model is **forms-based authentication**. In Spring Security provides rich and robust support for forms-based authentication models.

Now in the enterprise world, we often see integration with LDAP as a method of authentication. In **Spring Security provides native LDAP support as well as support for Microsoft Active Directory using the LDAP abstraction**. This is clearly one of the most common integration points in the world for Java based applications using Spring Security. Spring Security also has rich support for other external facing authentication providers that are much more common in an Internet based system.

Spring can support OAuth connect and OpenID as well as Jasig CAS for single sign on on authentication operations. Also, as you might expect from a Java world, there's also support for JAAS. Finally, Spring natively support Kerberos and SAML. We're going to go into those topics in this course. But if for some reason you find yourself needing to implement these protocols, note that Spring Security can take away some of the pain, not all of it, but some of it.







The **spring-security-core** project contains all the classes needed for core authentications and authorizations. Any application using spring security requires this class.

In addition to the core class, any custom configurations using XML or Java config requires the **spring-security-config** project.

Once you move into servlet API for web-based applications, you need **spring-security-web**.

The final core project that you will use as often as the core project itself, hopefully anyway, is spring-security-test, which provides the apparatus needed to adequately test applications that are using spring security.



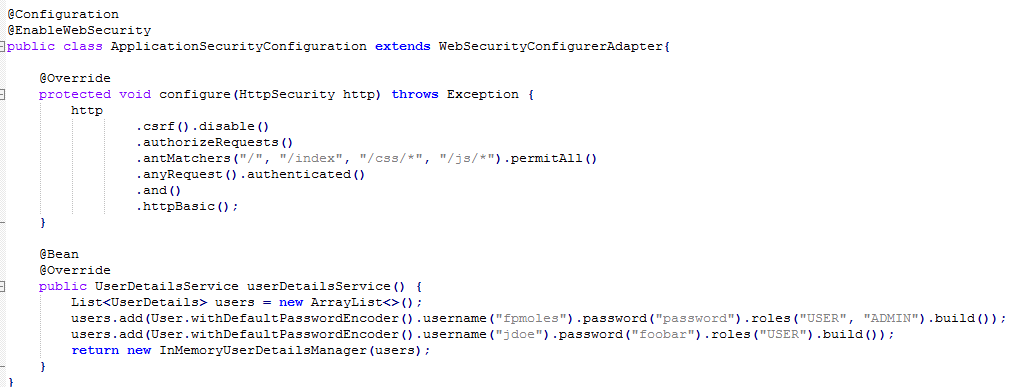
Form based authentication is the default behavior of Spring Security. Once it is added to project, it enables form-based authentication for the entire application. The default user name is “user”.

In Http builder the first thing we're going to do is CSRF dot disable to prevent that exploit. And now we're going to type in authorize request. So, this is where we're going to go in and configure how spring security is going to work. Now we're going to use an ant matcher, and that ant matcher is going to allow us to specify the urls that we want to allow.

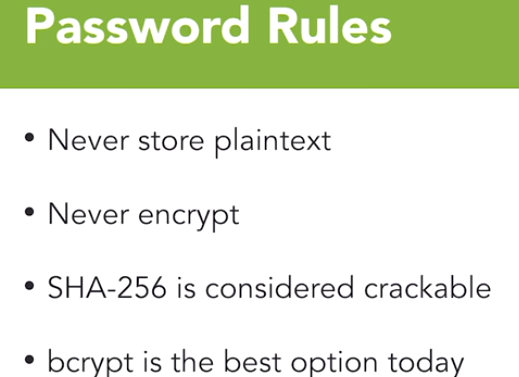
In this case, we're going to allow access to root, access to index, because both of those will respond to that root index page that we have. We also need to allow the css and the JavaScript file, or js, that are in static. So once these are done, we can now do a permit all. So now that we've specified where we want to do the permit, we need to tell spring security that anything else, any other request, we want to be authenticated.

Later, this is where we will mess with authorization as well, but for now, authenticated is good. And now we're going to give it a separate and command to let it know that I want to use http basic.

**In-Memory Authentication:**

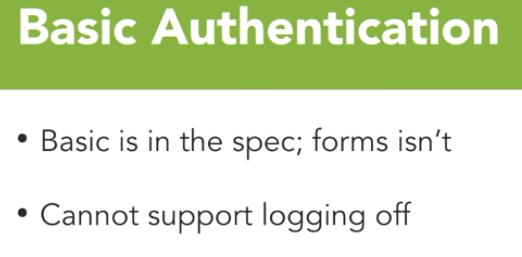


To add user and password.



When it comes to hashing passwords, SHA-256 seemed to be considered most universal. MD5 has been dead for a long time, so don't even consider it. SHA-256, however, is now considered crackable. It doesn't take that many GPUs to generate two hashes of the same value. Today, the best option that I can use is bcrypt. And the bcrypt libraries are built in to Spring Security, which makes them super easy for you to implement to make your code work and protect your customer's data.

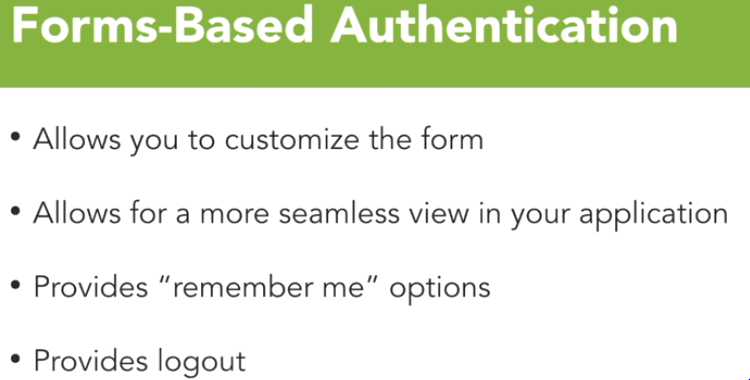
Hashing algorithm on an authentication routine to go too fast, because it allows your site to be brute-forced much easier. That little bit of delay, will slow down a bot dramatically. So, that's a feature of bcrypt. And quite frankly users are not going to be affected by that delay the way that a system will. That's all there is to getting bcrypt working in your application. You really have no reason not to user proper hashing when using Spring Security.



 One of the biggest issues with basic authentication is the inability to log off.

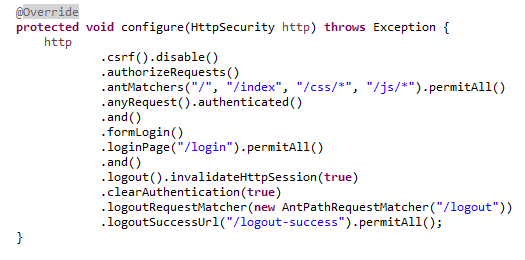
Everything is session based with basic authentication, and as such there's not a real unified way to handle logging off across all the browsers. Most of them don't even implement anything. Often, you will hear the base 60 form coding of the user name and password is a security flaw in the spec

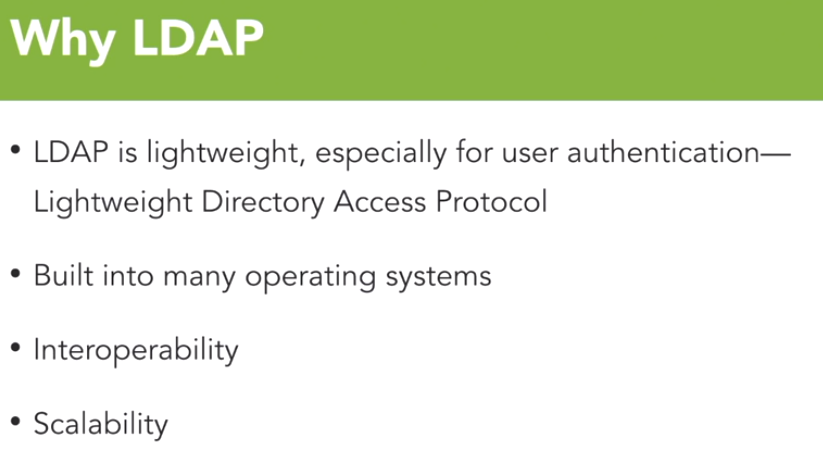
Using basic authentication also isn't very attractive. You get no ability to modify the login window either. And that causes many organizations heartburn.



With forms-based authentication you have complete control. Since you leverage a standard HTML form to pass the credentials to the back end, you can style that form as much or as little as you or your organization would like.

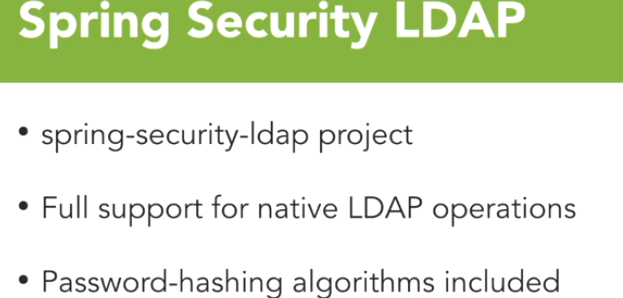
**To enable form-based login we need to do some change in configuration file:**

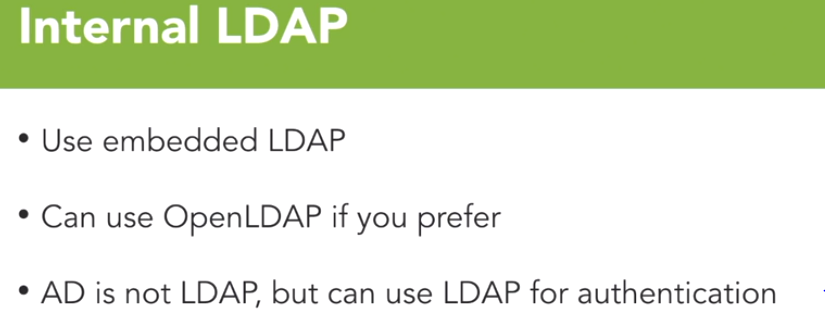


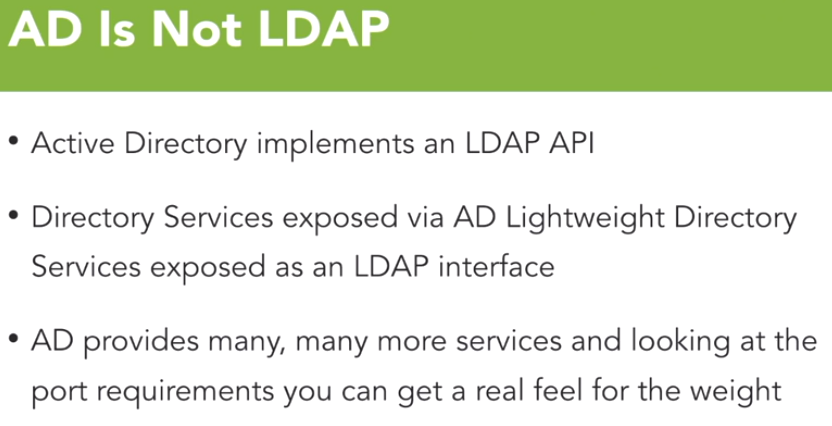


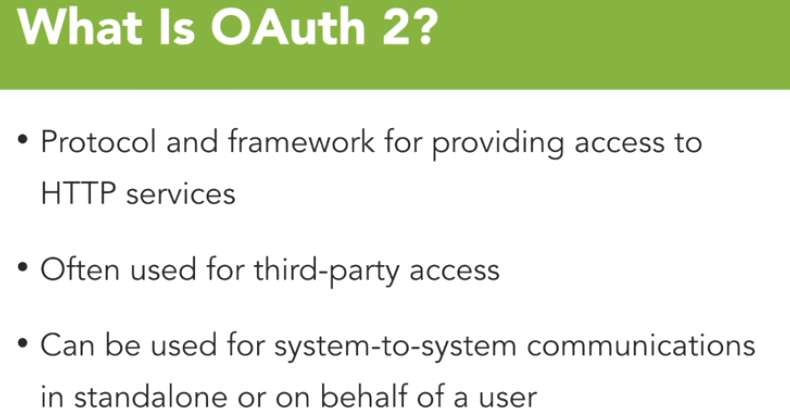
One of the most common ways to authenticate an application in an enterprise is using LDAP. Now, LDAP, or Lightweight Directory Access Protocol, is a lightweight user authentication directory structure. It's built in natively into many operating systems and of course, there's stand-alone implementations as well. One of the biggest benefits of LDAP is the fact that it's interoperable among many different operating systems and components.

Because LDAP is lightweight, it's also highly scalable, which, again, is another benefit when you're running distributed systems across the globe and all those enterprise systems need to share the same directory structure









OAuth 2 is a protocol as well as a framework for providing access to HTTP services. Now those services can be traditional web services as well as web applications

It's often used for third-party access, so if you use social media and you grant an application access to your Facebook profile, for instance, that's done through OAuth 2. Google provides the same sort of structure with its frameworks. It's so common that you may not even realize you’re doing it, but OAuth 2 really is all over the place. It also can be used for system-to-system communications in standalone mode or on behalf of another user. And from my perspective, this is the most common use case that I use because we secure all our web services calls with OAuth 2 between client and server on purpose to make sure that we're protecting the data as needed.

