

JSecurity
April 16, 2008

Charlotte
Java
Users
Group



Introduction

- Les Hazlewood
 - JSecurity Project Founder
 - les@hazlewood.com
 - <http://www.leshazlewood.com>



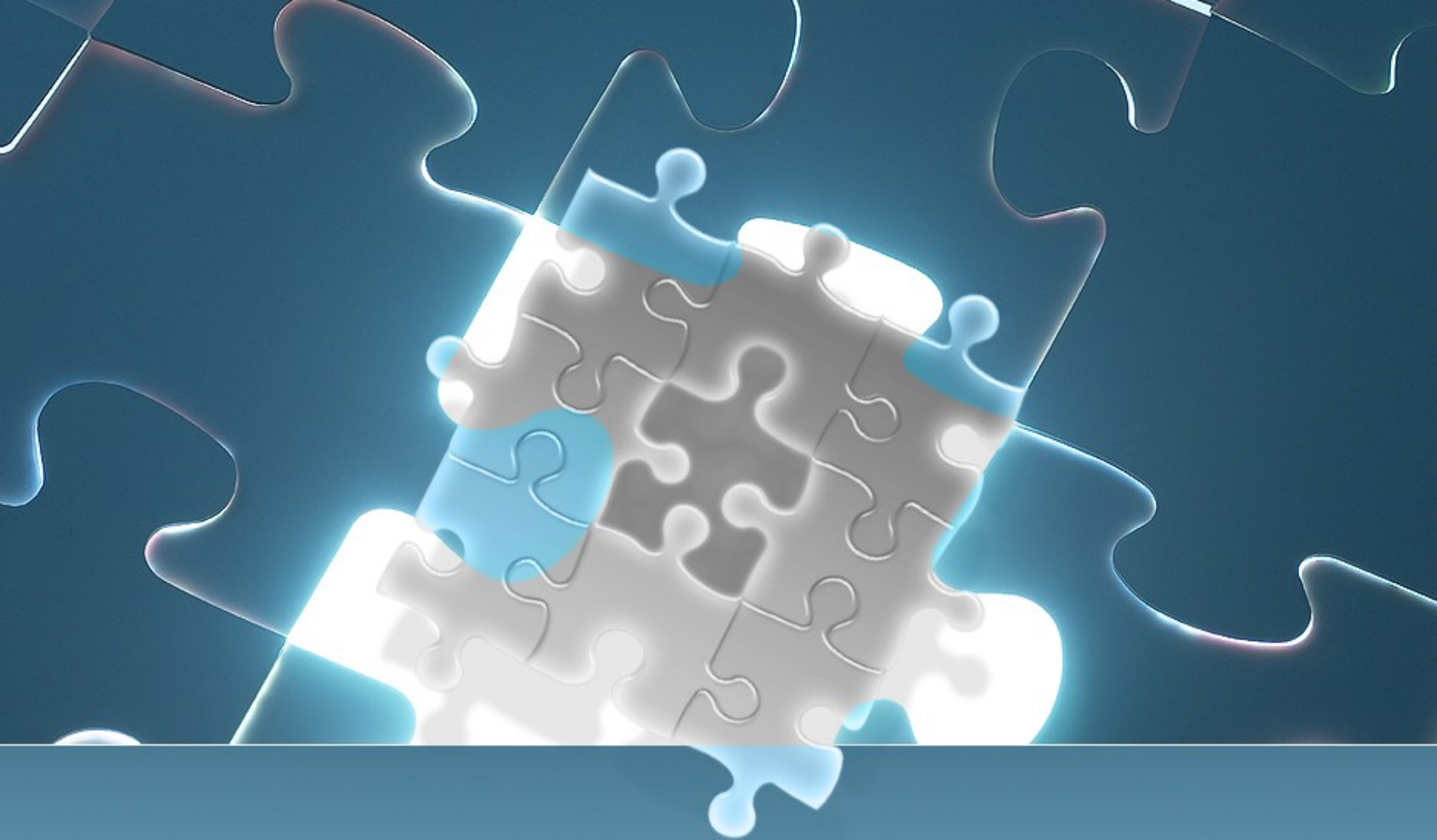
Presentation Format

- Topics areas introduced
- Selected issues presented
- Brief Demo
- Q&A



Tonight's Topics

- Project Background
- Quick Terminology
- Authentication (logging in)
- Authorization (access control)
- Enterprise Session Management
- Cryptography
- Web Support
- Demo
- Q&A



Project Background



Project Background

- Simplify or replace JAAS
- Change user/role/permission assignments *dynamically*, during *runtime*
- Access session data from multiple clients & servers
- Single Sign-On
- POJO-based: free from J2EE & container-specific APIs



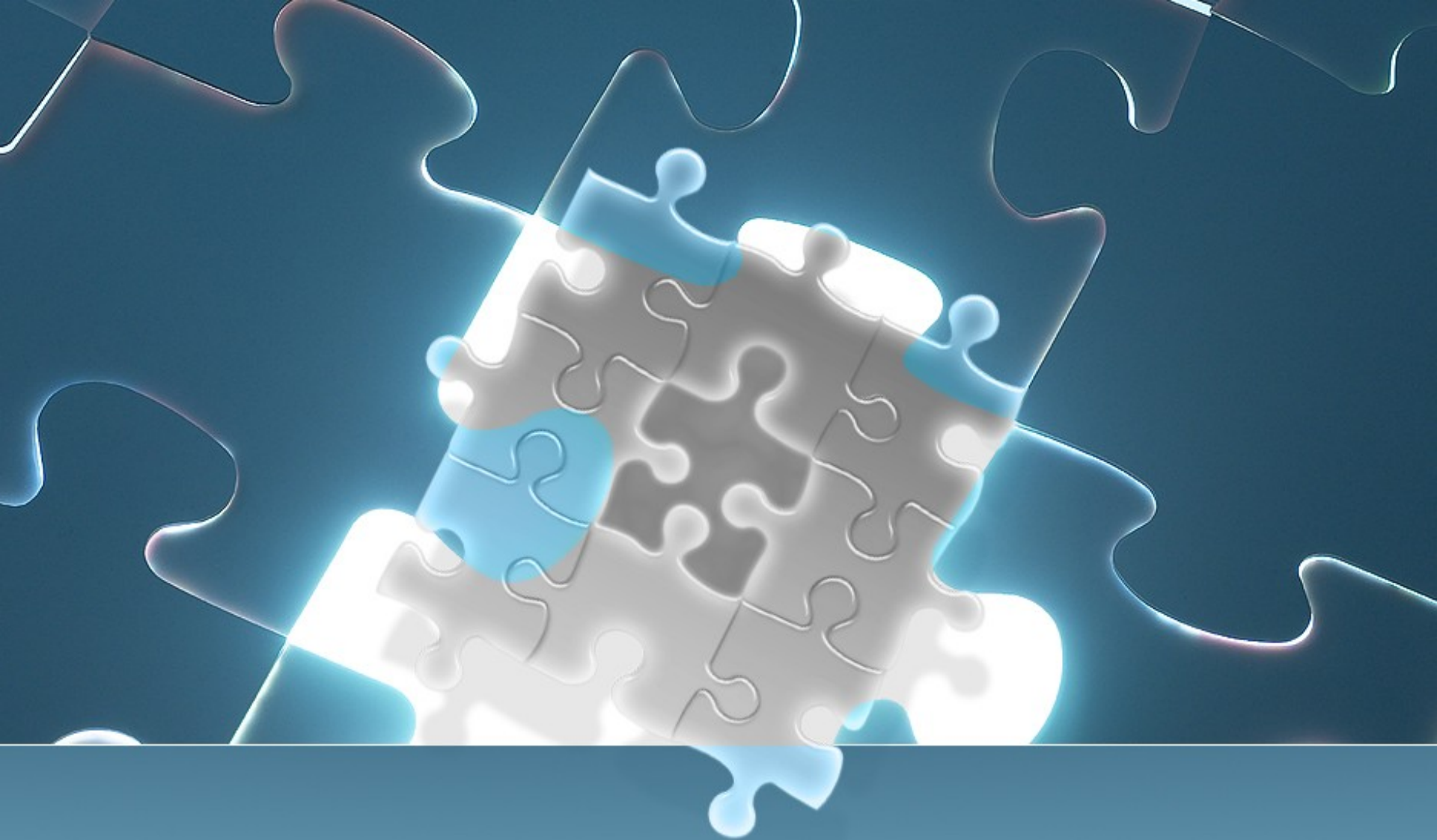
Quick Terminology

- **Realm:** A component that can access an application's security components such as users, roles, permissions. Usually datasource-specific.
- **Subject:** A security-specific representation of an application user (person, 3rd party process, etc).



Quick Terminology (cont'd)

- Principals: a Subject's identifying attributes such as username, user id, Social Security #, etc.
- Credentials: values known only to a related Subject, used to verify identity.
- Roles, Permission and Session concepts covered later



Authentication



Authentication

The act of verifying you are who you say you are.

1. Collect principals & credentials
2. Submit them to the system
3. If submitted credentials match those in the system, continue access.
4. If submitted credentials don't match, allow retry or block access.



Authentication – Step 1

1. Collect Principals and Credentials

```
//Example using most common scenario:  
//String username and password. Acquire in  
//system-specific manner (HTTP request, GUI, etc)  
  
UsernamePasswordToken token =  
    new UsernamePasswordToken( username, password );  
  
//"Remember Me" built-in, just do this:  
token.setRememberMe( true );
```



Authentication – Steps 2-4

2 - 4. Submit then allow or disallow.

```
Subject currentUser = SecurityUtils.getSubject();
try {
    currentUser.login(token);
} catch ( UnknownAccountException uae ) {
    //no account for the submitted username - retry?
} catch ( IncorrectCredentialsException ice ) {
    //submitted password was incorrect - retry?
} catch ( LockedAccountException lae ) {
    //account currently locked - unusable - nice error msg.
} catch ( ExcessiveAttemptsException eae ) {
    //too many unsuccessful login accounts. Lock it?
} ... catch more or your own custom exceptions ...
} catch ( AuthenticationException ae ) {
    //unexpected error?
}
//No problems, show authenticated view..
```



Authentication (cont'd)

Subject – interface w/ built-in login:

```
public void login( AuthenticationToken token)
    throws AuthenticationException;
```

Executes log-in attempt. Implementation of this interface usually delegates to a system-wide `Authenticator` which supports PAM.

If unsuccessful, handle in application-specific manner.

If successful, the Subject is automatically associated with account data.

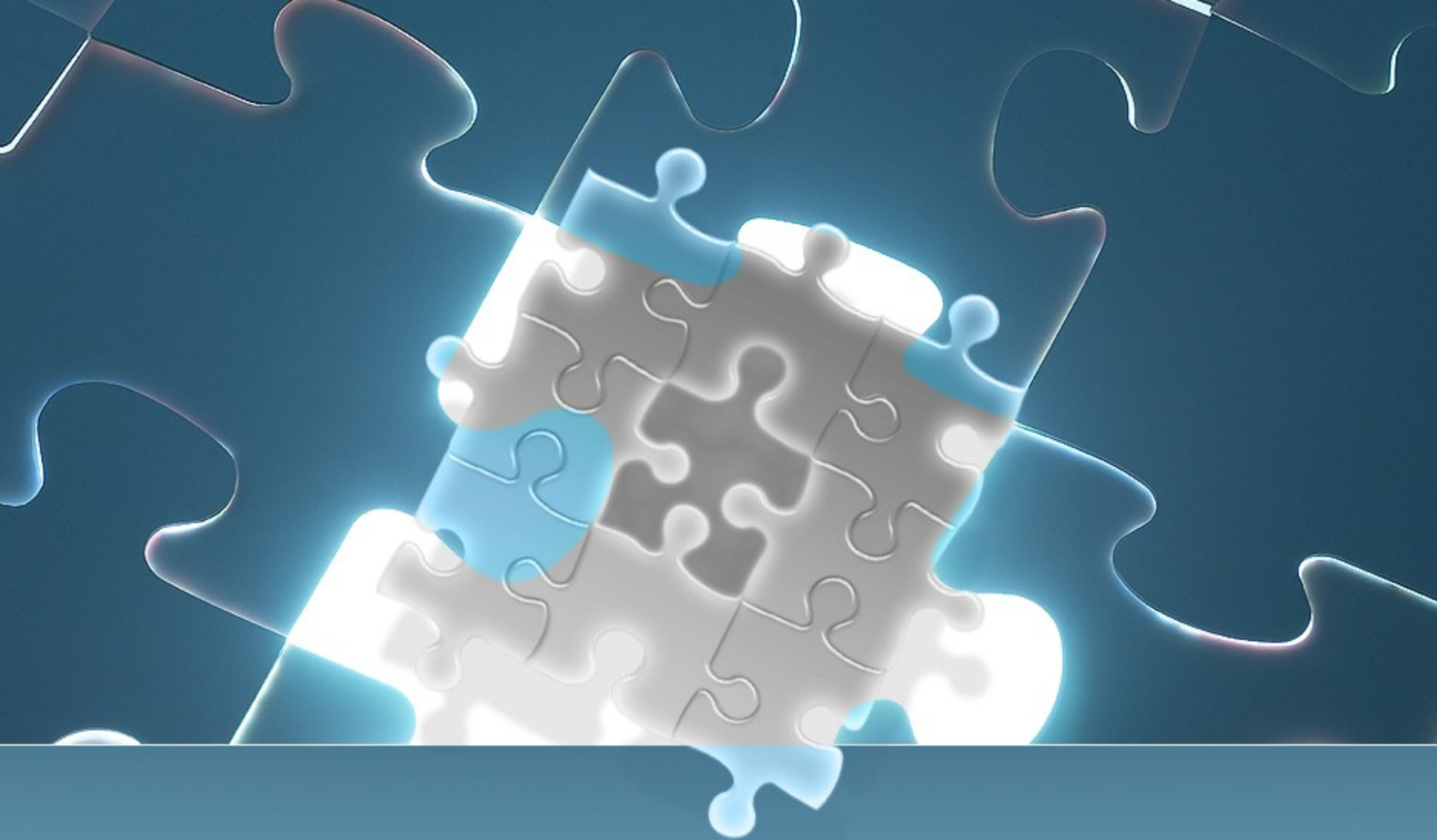
Other Subject methods are part of the Authorization presentation.



Authentication (cont'd)

Default `Authenticator` implementation in the `Jsecurity` is the `ModularAuthenticator`.

- Implements PAM (Pluggable Authentication Module) functionality.
- Utilizes a configured set of `Realms`
- Out of the box `Realm` implementations for Memory (RAM), LDAP, ActiveDirectory, JDBC, and others.
- Create your own for application-specific data access (e.g. Hibernate or JPA)



Authorization



Authorization

Once identity is validated, authorization is the process of determining access control – i.e. “who can do what”, or “who can do which actions”.

- Generally assumes Realm-based approach: an aggregation of users (a.k.a. Subjects), user groups, roles, and permissions.
- JSecurity can work with any data model, even if yours doesn't coincide with these assumptions



Permissions

- The “what” of the application.
- Most atomic element of a security system.
- Describes behavior for a *type* of object
- Does not maintain information about “who” can perform such behavior.

Example: A `PrinterPermission` instance defines if a printer can be accessed or not. It does *not* associate who can or cannot access that printer.



Roles

- A named collection of Permissions
- Allows behavior aggregation
- Enables dynamic (runtime) alteration of user abilities.
- Most similar to what we see in Windows or Unix user “groups”.

Example: An “Administrator” role might have the AllPermission.

- Still does not define “who” can do what - just organize multiple abilities.



Users

- The “who” of the application.
- What each user can do is defined by their association with Roles.

Example: A user “has a” collection of Roles. If one or more of those roles “has a” PrinterPermission, then that user can print to a specific printer.



Authorization

Multiple means of checking access control:

- Programmatically
- JDK 1.5 annotations
- JSP/GSP TagLibs



Programmatic Authorization

Checking for a particular Role:

```
//get the current Subject executing
//the code
Subject currentUser = SecurityUtils.getSubject();

if ( currentUser.hasRole( "administrator" ) {
    //do one thing (show a special button?)
} else {
    //do something else (don't show the button?)
}
```



Programmatic Authorization

Checking for a particular Permission:

```
//get the current Subject executing
//the code
Subject currentUser = SecurityUtils.getSubject();

Permission printPermission = new
    PrinterPermission( "laserjet3000n", "print" );
If ( currentUser.isPermitted( printPermission ) {
    //do one thing (show the print button?)
} else {
    //do something else (don't show the button?)
}
```



Programmatic Authorization

Permission checks (String based)

Same permission check as in previous slide:

```
if ( currentUser.isPermitted(
    "printer:print:laserjet3000n") {
    //do one thing (show the print button?)
} else {
    //do something else (don't show the button?)
}
```



Annotation Authorization

Role(s) check:

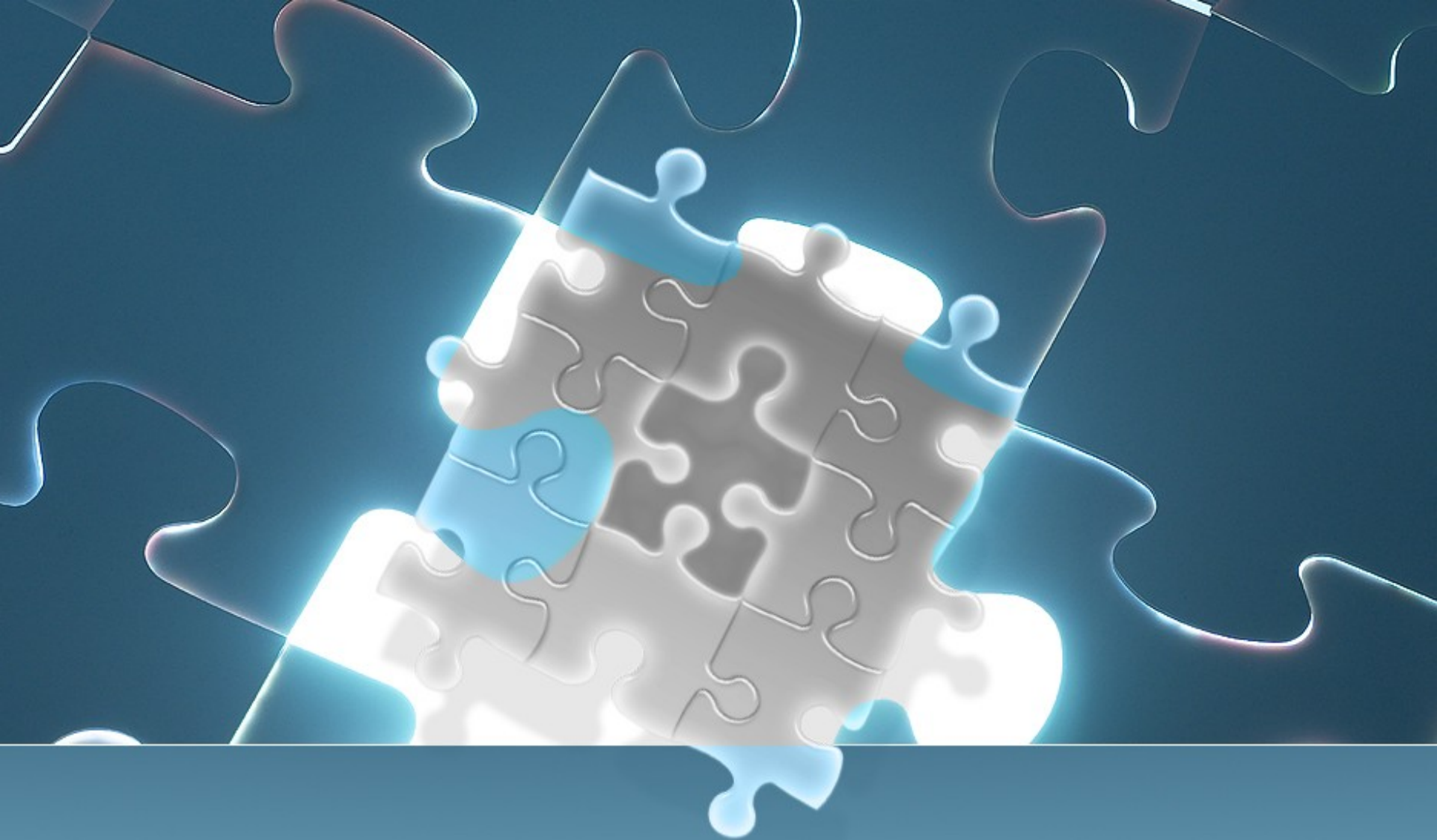
```
//next code block will throw an
//AuthorizationException if the caller doesn't
//have the 'teller' role:
@RequiresRoles( "teller" )
public void openAccount( Account acct ) {
    //do something in here that only a teller
    //should do
}
```



Annotation Authorization

Permission(s) check:

```
//Will throw an AuthorizationException if none
//of the caller's roles imply the Account
//'create' permission
@RequiresPermissions("account:create")
public void openAccount( Account acct ) {
    //create the account
}
```



Session Management



Session Management

- Impetus for creation – Heterogeneous client access
- Simplified Configuration
- POJO/J2SE based (IoC friendly)
- Not bound by client or specification protocol.
- Can be used in Single-Sign On implementations



Session Management (cont'd)

- Temporal-based record (start, stop, last access times).
- Event-trigger support
- IP address correlation (if needed)
- Inactivity and expiration support (touch() method)
- Retains HttpSession methods you're comfortable with (get/set/remove Attribute methods)



Session Management (cont'd)

- Session Management is typically transparent to application developers
- Interceptor/AOP mechanisms to create, associate, and maintain Sessions
- Event support enables transparent activity logging.
- Session Data (Attributes and metadata) can be stored anywhere – in a database, on the file system, in memory, a distributed cache, etc. (DAO to back-end store).



Session Management (cont'd)

Acquiring and creating sessions. For convenience, things work just like the HttpSession:

```
//guarantee a session exists (will  
//create one if it does not)  
subject.getSession();
```

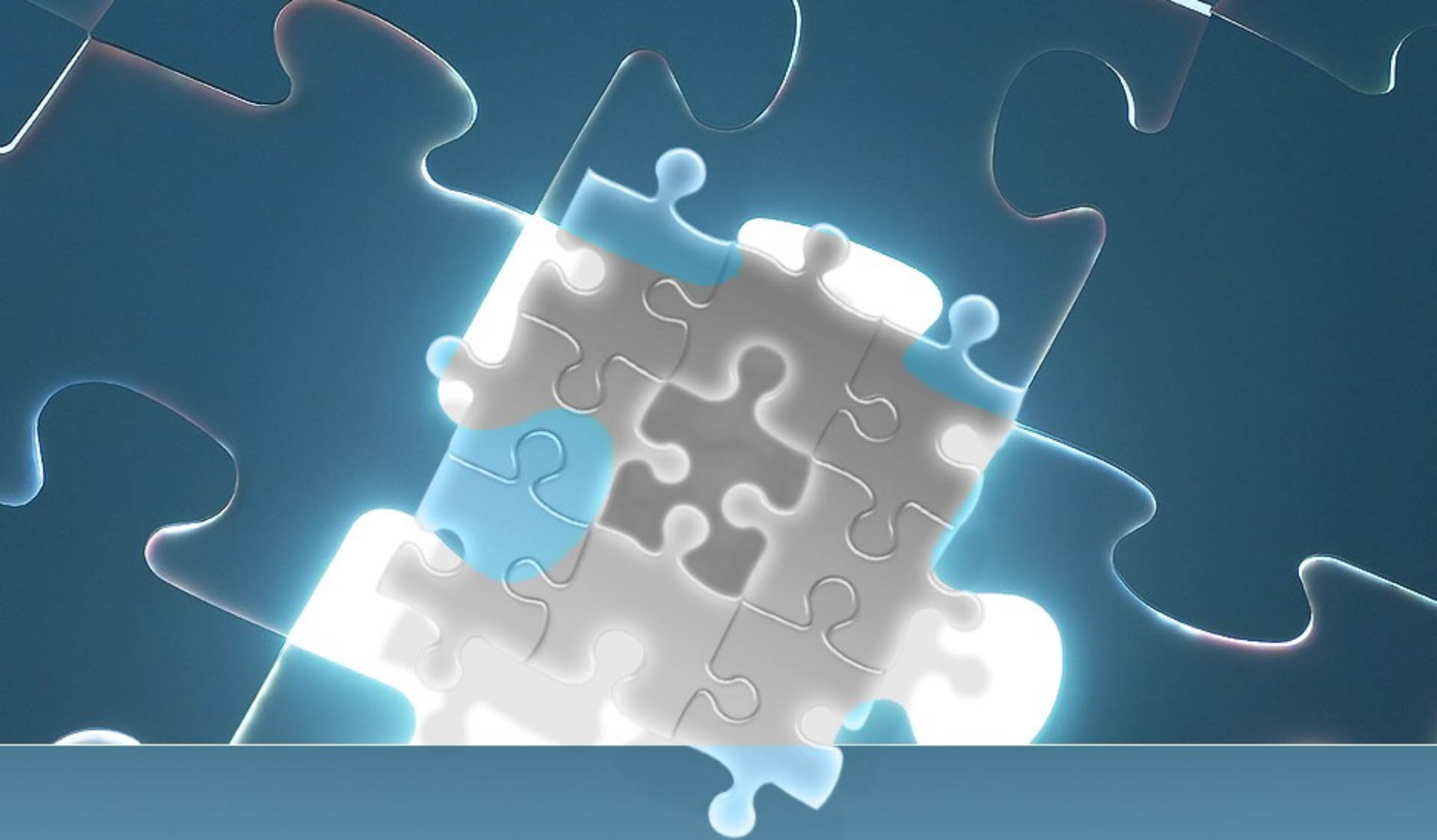
```
//get a session if it exists. If it does  
//not, null will be returned  
subject.getSession(false);
```



Session Management (cont'd)

But note! JSecurity Sessions do **not** require a HTTP environment!

- Works in any tier (server-side Service, web MVC client, Swing client, etc).
- Transparent use in web tier – no need to change your existing HttpSession-based code.
- Fully clusterable with very easy set-up: TerraCotta, GigaSpaces, etc.
- Just Pojos.



Cryptography



Cryptography

Maintains JSecurity's ideals of simplicity:

- Interface-driven, POJO based
- Simplified wrapper over the very complex JCE infrastructure.
- “Object Orientifies” cryptography concepts, whereas JCE has a more procedural language feel.
- Easier to understand API (ciphers, hashes, etc).



Ciphers

A Cipher is a cryptographic algorithm that encrypts and decrypts data based on public and/or private keys.

- Symmetric Cipher uses the same (or trivially similar) Key during both encryption and decryption.
- An Asymmetric Cipher uses one Key during encryption and a different Key during decryption.



Ciphers

JSecurity's Cipher interface:

```
public interface Cipher {  
    byte[] encrypt( byte[] raw, Key encKey );  
    byte[] decrypt( byte[] encrypted, Key decKey);  
}
```

- Initial default implementations exist (BlowfishCipher et. al.)
- Encrypt user identities, such as RememberMe cookies, and more.



Hashes

A cryptographic Hash (aka Message Digest) is a one-way, irreversible conversion of an input source

- Commonly used for credentials (e.g. password) transformation for added security.
- Can be used on anything: Files, Streams, etc. - anything with an underlying byte array.



Hashes

JSecurity's Hash interface:

```
public interface Hash {  
    byte[] getBytes();  
    String toHex();  
    String toBase64();  
}
```

- Default implementations exist: MD5, SHA1, SHA-256, et. al.
- Encrypt user identities, such as RememberMe cookies, and more.

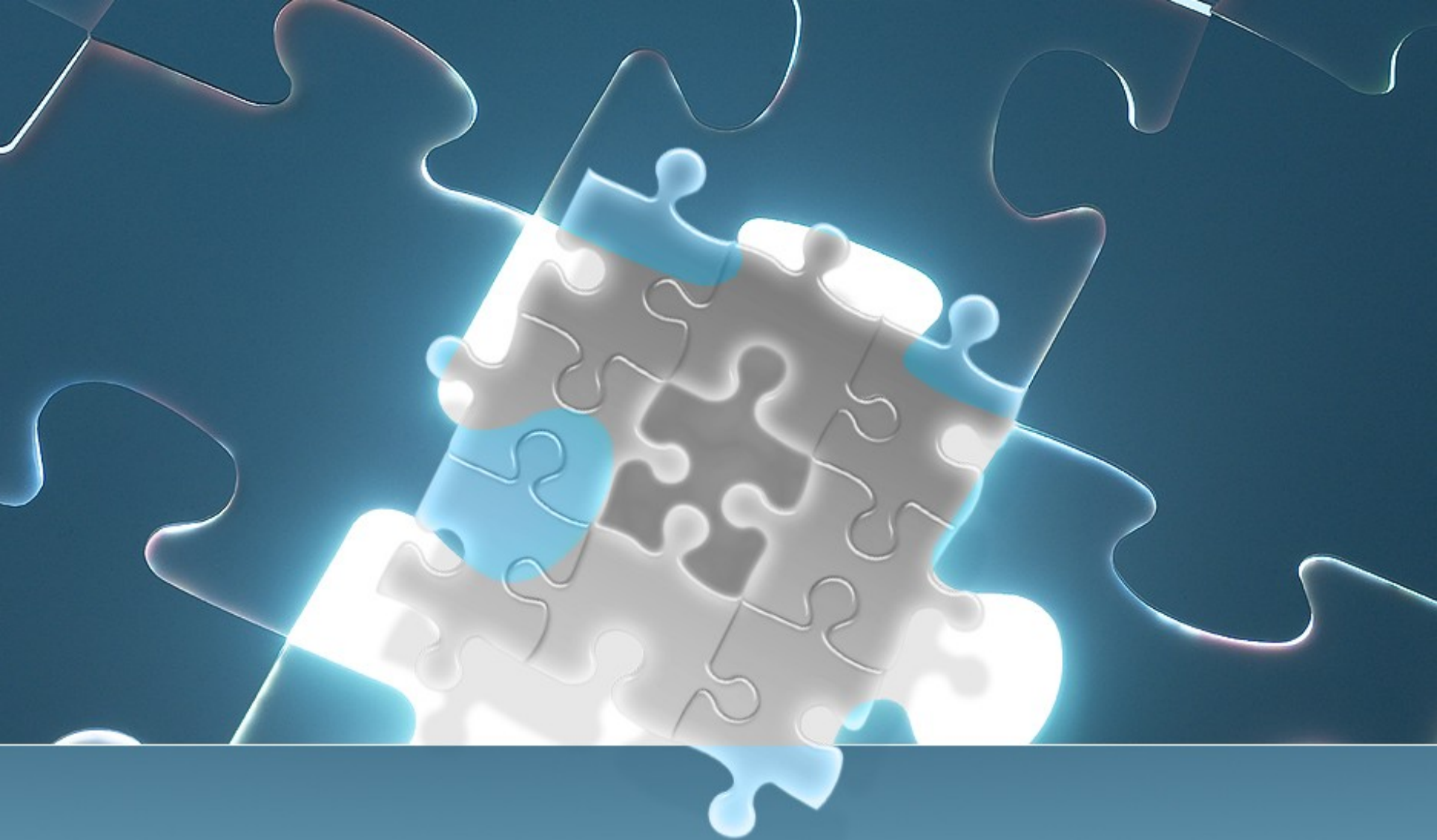


Hashes

Intuitive OO API

```
//some examples:  
new Md5Hash("blah").toHex();  
//File MD5 Hash value for checksum:  
new MD5Hash( aFile ).toHex();  
//store a password, but not in raw form:  
new Sha256(aPassword).toBase64();
```

- Cleaner OO API compared to procedural APIs like JDK MessageDigest or Commons Digest.
- Built in hex & base64 conversion
- Built-in support for Salts and repeated hashing



Web Support



web.xml

```
<filter>
  <filter-name>JSecurityFilter</filter-name>
  <filter-
    class>org.jsecurity.web.servlet.JSecurityFilter</filter-class>
  <init-param><param-name>config</param-name><param-value>
    [interceptors]
    authc.successUrl = /index.jsp

    [urls]
    /account/** = authc
    /remoting/** = authc, roles[b2bClient],
    perms[remote:invoke:"lan,wan"]
  </param-value></init-param>
</filter>

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



JSP TagLib Authorization

```
<%@ taglib prefix="jsec" uri=  
http://www.jsecurity.org/tags %>
```

```
<html>
```

```
<body>
```

```
  <jsec:hasRole name="administrator">
```

```
    <a href="manageUsers.jsp">Click here to  
manage users</a>
```

```
  </jsec:hasRole>
```

```
  <jsec:lacksRole name="administrator">
```

```
    No user admin for you!
```

```
  </jsec:lacksRole>
```

```
</body>
```

```
</html>
```



JSP TagLibs

```
<%@ taglib prefix="jsec" uri=  
    http://www.jsecurity.org/tags %>
```

```
<!-- Other tags: -->
```

```
<jsec:guest/>
```

```
<jsec:user/>
```

```
<jsec:principal/>
```

```
<jsec:hasRole/>
```

```
<jsec:lacksRole/>
```

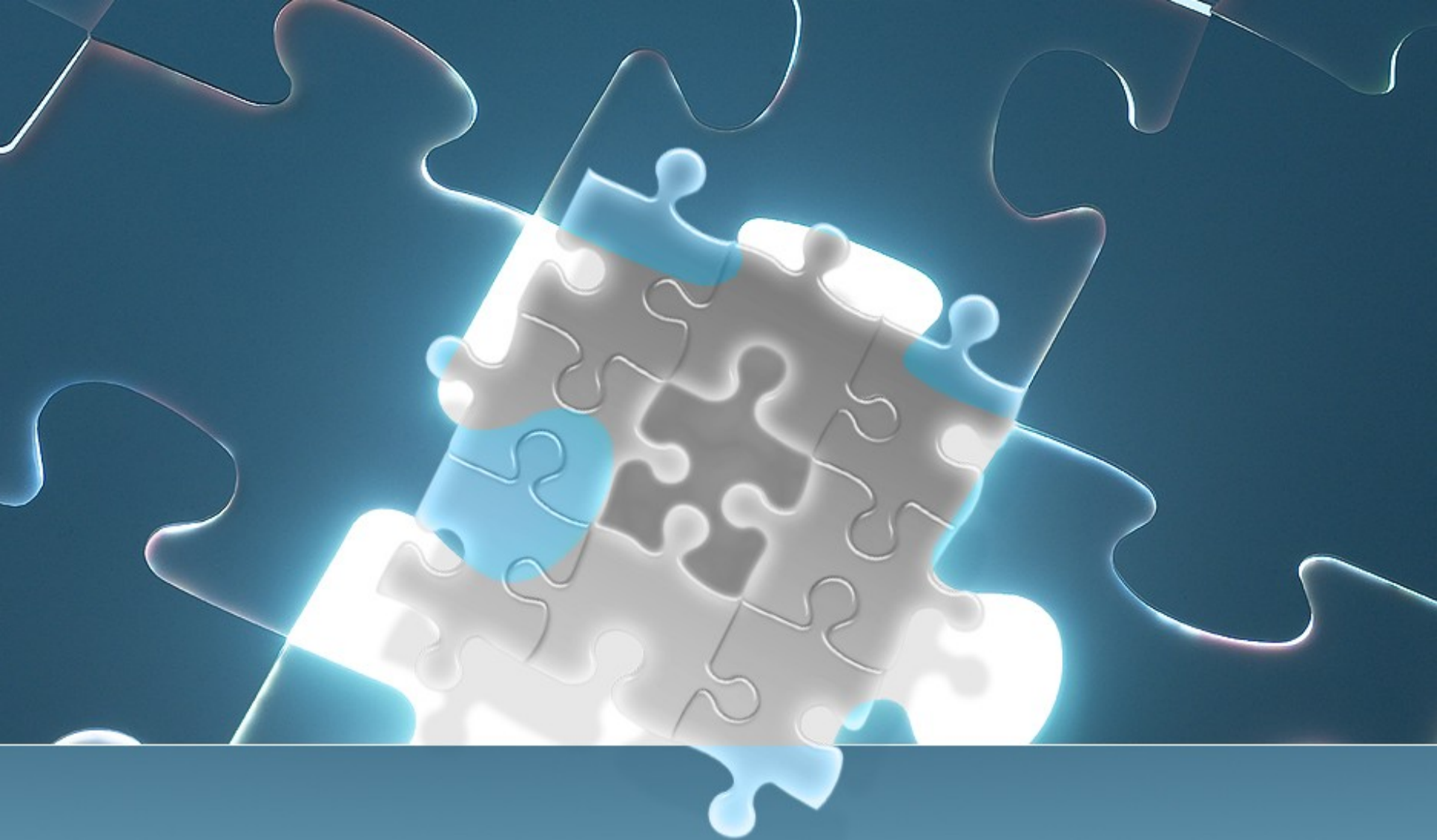
```
<jsec:hasAnyRoles/>
```

```
<jsec:hasPermission/>
```

```
<jsec:lacksPermission/>
```

```
<jsec:authenticated/>
```

```
<jsec:notAuthenticated/>
```



Logging-out



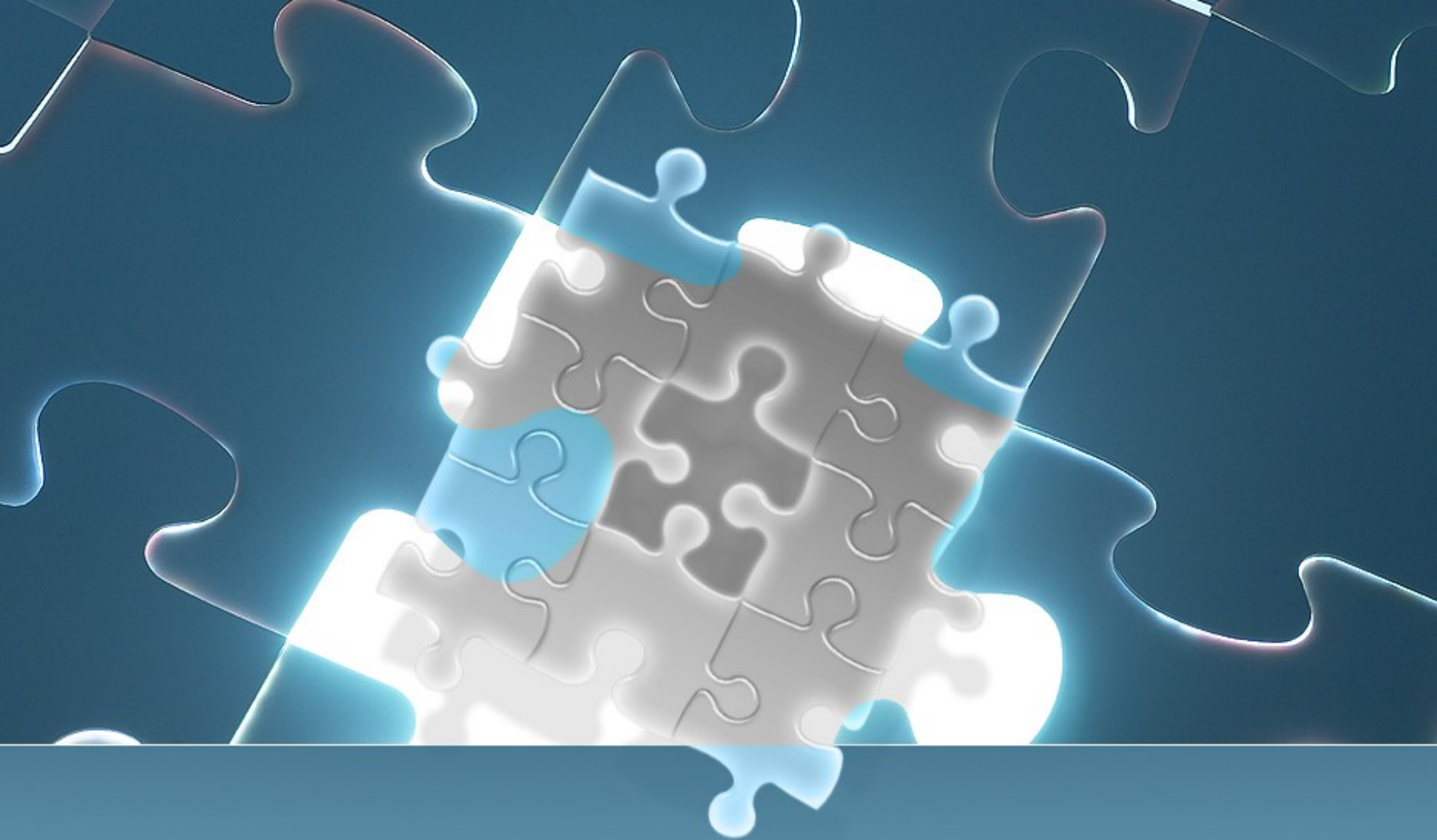
Logging-out

One method:

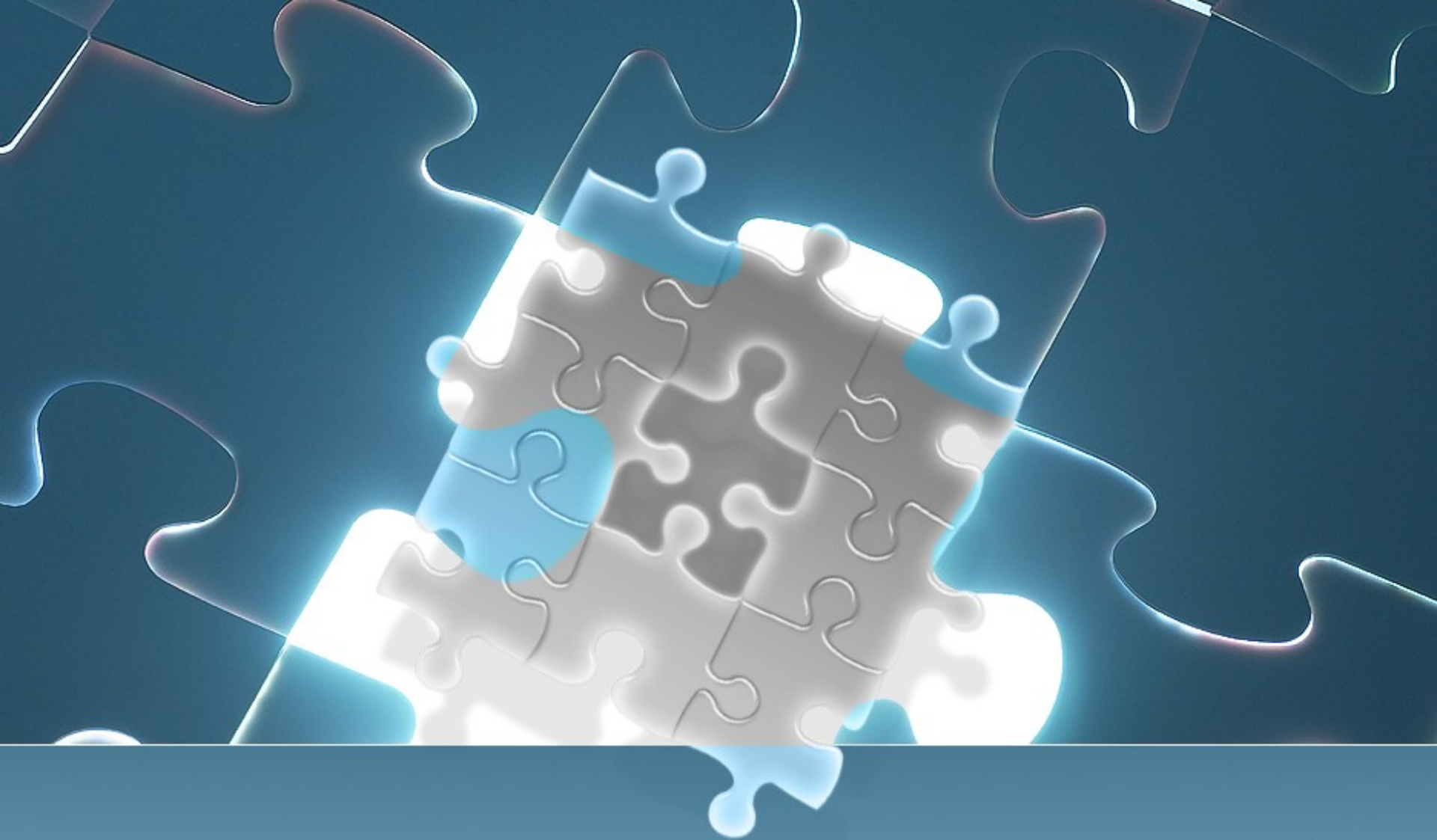
```
//Logs the user out, relinquishes account data, and  
//invalidates any Session  
Subject currentUser = SecurityUtils.getSubject();  
currentUser.logout();
```

App-specific log-out logic is can be executed in any number of ways:

- Execute explicitly either preceding or after this call
- Listen for the StoppedSessionEvent and trigger the logic based on the event
- Both/Either.



Demo



Questions?