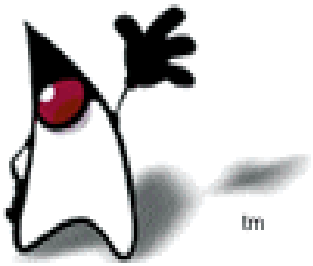
Web Enterprise Application Programming



Color Objects for the RESTful API

Color Object Conversions

- We have already seen the notion of a JSF Converter, for example for Color
  - Color Object converted to text for display/edit on JSF page
    - #RRGGBB or other formats that a user can edit
    - Similar to dates, we need to agree with our user on the format of the string
  - Text String converted to Color object when JSF page submitted
    - Need to parse the string, convert to rgb values, create color object with values

- For the RESTful API, we need to convert Color objects to JSON/XML and back:
  - JSON: {"x": 4, "y": 7, "color": {"red": 34, "green": 127, "blue": 45}, "dx": -3 …}
  - XML: <sprite><x>4</x><y>7</y><color><red>34</red><green>127</green><blue>45</blue></color><dx>-3</dx>…</sprite>

XML conversions

```
@Entity
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Sprite implements Serializable {
…
…
    @XmlElement
    @XmlJavaTypeAdapter(ColorAdapter.class)
    @JsonbTypeDeserializer(JsonColorDeserializer.class)
    @JsonbTypeSerializer(JsonColorSerializer.class)
    @Column
    private Color color = Color.BLUE;
```

JSON conversions

```
@Entity
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Sprite implements Serializable {
…
…
    @XmlElement
    @XmlJavaTypeAdapter(ColorAdapter.class)
    @JsonbTypeDeserializer(JsonColorDeserializer.class)
    @JsonbTypeSerializer(JsonColorSerializer.class)
    @Column
    private Color color = Color.BLUE;
```

For XML: ColorAdapter.java

```
public class ColorAdapter extends XmlAdapter<ColorAdapter.ColorValueType, Color> {
    @Override
    public Color unmarshal(ColorValueType v) throws Exception {
        return new Color(v.red, v.green, v.blue);
    }
    @Override
    public ColorValueType marshal(Color v) throws Exception {
        return new ColorValueType(v.getRed(), v.getRed(), v.getBlue());
    }
    @XmlAccessorType(XmlAccessType.FIELD)
    public static class ColorValueType {
        private int red;
        private int green;
        private int blue;
        public ColorValueType() {
        }
        public ColorValueType(int red, int green, int blue) {
            this.red = red;
            this.green = green;
            this.blue = blue;
        }
    }
}
```

For JSON: JsonColorDeserializer

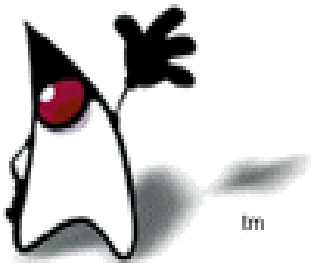
```
public class JsonColorDeserializer implements JsonbDeserializer<Color> {
    @Override
    public Color deserialize(javax.json.stream.JsonParser parser, javax.json.bind.serializer.DeserializationContext ctx,
java.lang.reflect.Type rtType) {
        String keyname = "";  int value = 0; int red = 0; int green = 0; int blue = 0;
        while (parser.hasNext()) {
            Event event = parser.next();
            switch (event) {
                case KEY_NAME: {
                    keyname = parser.getString();
                    break;
                }
                case VALUE_NUMBER: {
                    value = parser.getInt();
                    if (keyname.equals("red")) red = value;
                    else if (keyname.equals("green")) green = value;
                    else if (keyname.equals("blue")) blue = value;
                    break;
                }
            }
        }
        return new Color(red,green,blue);
    }
}
```

For JSON: JsonColorSerializer

```
public class JsonColorSerializer implements JsonbSerializer<Color>  {

    @Override
    public void serialize(Color c, JsonGenerator jg, SerializationContext ctx) {
        jg.writeStartObject();
        jg.write("red", c.getRed());
        jg.write("green", c.getGreen());
        jg.write("blue", c.getBlue());
        jg.writeEnd();
    }
}
```

Web Enterprise Application Programming



Security: User Accounts

Identification

Authentication

Authorization

Web App Accounts

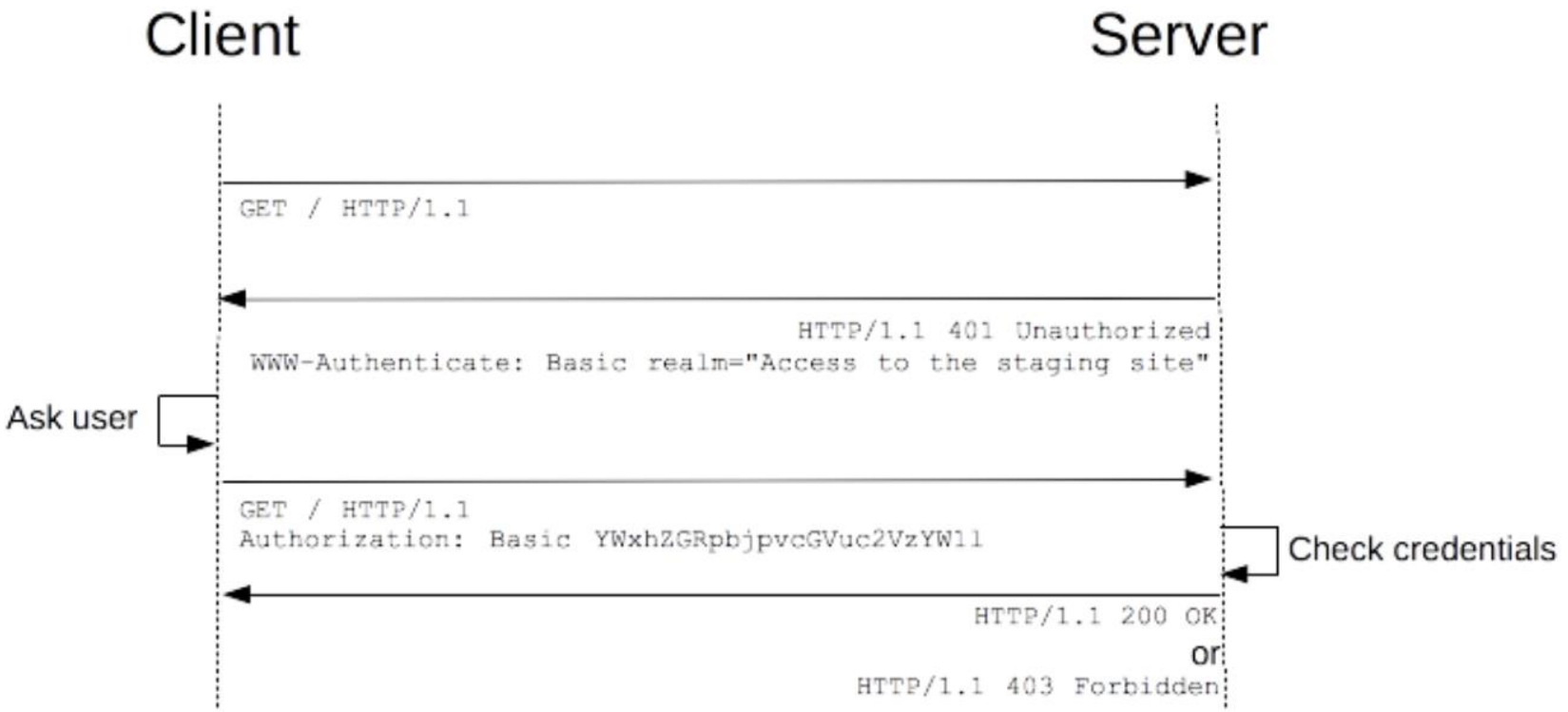
- Normally a web application will need to restrict and control access to resources

- Identification: distinguishing between users
- Authentication: identity is proved
- Authorization: authorized identity is granted access to resources

- The HTTP protocol itself supports various Authentication mechanisms
- The following two slides are based on:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

HTTP Authentication



HTTP Authentication

- In the case of a "Basic" authentication like shown in the figure, the exchange **must** happen over an HTTPS (TLS) connection to be secure.
- The challenge and response flow works like this:

1. The server responds to a client with a 401 (Unauthorized) response status and provides information on how to authorize with a WWW-Authenticate response header containing at least one challenge.
2. A client that wants to authenticate itself with the server can then do so by including an Authorization request header with the credentials.
3. Usually a client (browser) will present a password prompt to the user and will then issue the request including the correct Authorization header.

Glassfish HTTP vs HTTPS

- HTTPS for Glassfish is set up on port 8181 by default
- This won't work with modern browsers because it's a self-signed certificate
- We will work on HTTP (port 8080), knowing that in a real-world situation, we need to arrange for a proper TLS/SSL certificate (which usually costs money)

JEE Security

- Application Layer Security
  - Component containers
  - What about data traveling across network?
- Transport Layer Security
  - TLS (was SSL)
  - TLS/SSL is necessary for security when using Basic Auth
    - Basic Auth does only base64 encoding of username/password (easily reversed)
    - We use Basic Auth without TLS/SSL in this course, but you would NEVER DO THAT IN PRODUCTION or with a publicly accessible system
  - https:// urls
  - Involves on trusted Certificate Authorities
  - Not sufficient on its own – need Application Layer Security also
- Message-Layer Security (not in this course)
  - SOAP related

Securing JEE Web Apps

- The Java EE Security API specification is on the JCP Ballot as JSR 375, for inclusion in Java EE 8.
- The reference implementation (RI) of JSR 375 is called Soteria.
- Before discussing JSR 375, we read some background information (covered by previous JEE Specifications):

https://docs.oracle.com/javaee/7/tutorial/security-intro005.htm

JEE 8 Security API

- What does it provide us?

- **@BasicAuthenticationMechanismDefinition**
- **@FormAuthenticationMechanismDefinition**
- @CustomFormAuthenticationMechanismDefinition
- HttpAuthenticationMechanism interface:
  - public AuthenticationStatus validateRequest(…)
- @ServletSecurity
- **IdentityStore interface**
  - **@DatabaseIdentityStoreDefinition**
  - @LdapIdentityStoreDefinition
- SecurityContext interface

Login Security: users

- Login security is based on Realms, Users, Groups, and Roles
- Usernames identify users, and users are grouped together in groups
  - Often the user will pick their username, and we set their group programmatically (as “guest” or “regular” for example)
- A Role is defined by specifying a set of users and groups (by default, each group is a role, which is the default role mapping)
- Username/Password database
  - OLD WAY: We set up a Realm (jdbc) which involves specifying where the system can retrieve usernames, passwords, and groups
  - NEW WAY: (Soteria) We set up an IdentityStore (DatabaseIdentityStoreDefinition) for usernames, passwords, groups
- At runtime, when a user browses to a path covered by a security constraint, the system (with Basic and Form) will check username, password, and group/role

Login Security: resources

- Resources are protected by requiring the user to be in a certain role
- JSF Pages: We set security constraints (in web.xml) on URL path patterns (for example, we might specify user must be in Role “admin” to access URLS with path ”/admin/*”
- RESTful API Resources: We require Roles by annotating the classes and methods corresponding to @PATH resources

Realms

- A **realm** is a complete database of users and groups identified as valid users of one or more applications and controlled by the same authentication policy.
- The Java EE server authentication service can govern users in multiple realms.

Realms cont'd

- `file` Realm: user credentials are stored locally in a `keyfile`
  - `file` realm has users added using Glassfish Admin Console
  - See glassfish->Configurations->Security->Realms->file
- `certificate` Realm: server stores user credentials in a certificate database
  - X.509 certificates
- `admin-realm` Realm
  - Also a file realm, for administrators of the Glassfish server
- `jdbc` Realm (old way): usernames, passwords, groups stored in a database accessed with JDBC
  - Enables us to add users programmatically at runtime
- NEW WAY `@DatabaseIdentityStoreDefinition`
  - Enables us to add users programatically at runtime

Groups

- Can be used to group users that have traits in common
- Cannot be used in Certificate Realm
- A group is designated for the whole Glassfish Server

Roles

- A Role entails permission to access a set of resources
- Users and groups are mapped to roles (giving those users permissions)
  - Done in `glassfish-web.xml (example below)`
- The default Role mapping is Groupname = Rolename
  - In this course, we'll use the default mapping
- Example from a glassfish-web.xml:

```
<glassfish-web-app> ...
  <security-role-mapping>
    <role-name>Mascot</role-name>
    <principal-name>Duke</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>Admin</role-name>
    <group-name>Director</group-name>
  </security-role-mapping> ...
</glassfish-web-app>
```

Basic and Form Authentication

- In web.xml, we can specify that Basic or Form Authentication should be used
  - Basic: browser's built-in username/password dialog is used
  - Form: programmer specifies to use a specific login page and error page
    - Example html code for a login page:

```
<form action="j_security_check" method="post">
   <input name="j_username" type="text"/>
   <input name="j_password" type="password"/>
   <input type="submit" />
</form>
```

Steps to secure logins to our apps

- Specify which paths of the webapp are protected, and by which realm
  - For JSF pages, we do this in web.xml based on URL patterns
  - For RESTful API, we annotate EntityFacadeREST.java
- Static Users: Use File Realm by adding Glassfish users manually at the Glassfish Admin Console->Configurations->Server Config->Security->Realms->File->Manage Users
- Dynamic Users (what source code file can we put this in?):

```
@DatabaseIdentityStoreDefinition(
   dataSourceLookup = "${'java:comp/DefaultDataSource'}",
   callerQuery = "#{'select password from app.contact where userId = ?'}",
   groupsQuery = "select groupname from app.contact where userId = ?",
   hashAlgorithm = PasswordHash.class,
   priority = 10
)
```

Steps to secure logins (cont'd)

- For JSF (put annotations in a Java file in our project, we'll use ApplicatonConfig.java):

  @BasicAuthenticationMechanismDefinition

  Or (in the following, substitute the real location and name of the forms)

```
@FormAuthenticationMechanismDefinition(
  loginToContinue = @LoginToContinue(
    loginPage = "/index.xhtml",
    errorPage = "/index.xhtml"))
```

- For RESTful API (in EntityFacadeREST.java):
  - @DeclareRoles
  - @RolesAllowed

Working with DataSources

- As long as you have completed and you understand all of Lab 2, you may use the database of your choice
- `java:comp/DefaultDataSource`
  - Uses Derby Database `sun-appserv-samples`
  - Username: app
  - Password: app
  - The username and password for sun-appserv-samples are in c:/glassfish5/glassfish/domains/config/domain1.xml
- In Lab2, we created a new DataSource for a non-Derby database of your choosing, using a Netbeans Wizard. Alternatively, we could create a new DataSource using code in `web.xml`

Classnames for MySQL versions

- When defining a datasource, be sure to use the right Driver classname, and DataSource classname for Derby (if you are using Derby) or the version of MySQL you are using (if you are using a version of MySQL)

- MySQL 8 Driver: com.mysql.cj.jdbc.Driver
- MySQL 8 Datasource: com.mysql.cj.jdbc.MysqlDataSource

- MariaDB, MySQL less than 8, Driver: com.mysql.jdbc.Driver
- MariaDB, MySQL less than 8, Datasource: com.mysql.jdbc.jdbc2.optional.MysqlDataSource

Define Datasource Derby (web.xml)

```
<data-source>
    <description>Todd's Tutoring data source.</description>
    <name>java:global/TgkDataSource</name>
    <class-name>org.apache.derby.jdbc.ClientDataSource</class-name>
    <server-name>localhost</server-name>
    <port-number>1527</port-number>
    <database-name>sun-appserv-samples</database-name>
    <user>app</user>
    <password>app</password>
    <property>
      <name>connectionAttributes</name>
      <value>;create=true</value>
    </property>
  </data-source>
```

Define DataSource MySQL < 8.0 or MariaDB (web.xml)

```
<data-source>
    <description>Todd's MariaDB data source.</description>
    <name>java:global/MySQLDataSource</name>
    <class-name>com.mysql.jdbc.jdbc2.optional.MysqlDataSource</class-name>
    <server-name>192.168.124.134</server-name>
    <port-number>3306</port-number>
    <database-name>myusers</database-name>
    <user>tgk</user>
    <password>tgkpass</password>
    <property>
       <name>connectionAttributes</name>
       <value>;create=true</value>
    </property>
</data-source>
```

Define DataSource MySQL 8.0 (web.xml)

```
<data-source>
    <description>Todd's MySQL8 data source.</description>
    <name>java:global/MySQLDataSource</name>
    <class-name> com.mysql.cj.jdbc.MysqlDataSource</class-name>
    <server-name>192.168.124.134</server-name>
    <port-number>3306</port-number>
    <database-name>myusers</database-name>
    <user>tgk</user>
    <password>tgkpass</password>
    <property>
       <name>connectionAttributes</name>
       <value>;create=true</value>
    </property>
</data-source>
```

DataSource Annotation

- Alternatively to web.xml, we can use the @DataSource annotation in a Java file to define the datasource (useSSL=false for MySQL8):

```
@DataSourceDefinition(
   name = "java:global/ExampleDS",
   className = "org.apache.derby.jdbc.ClientDataSource",
   portNumber = 1527,
   serverName = "localhost",
   databaseName = "exampleDB",
   user = "examples",
   password = "examples",
   properties={"create=true"})
```

JSF Pages: Add Constraint to paths

- Edit web.xml and select security tab; or put the following in web.xml

```
<security-constraint>
    <display-name>AdminConstraint</display-name>
    <web-resource-collection>
      <web-resource-name>moviedbadmin</web-resource-name>
      <description>Administration area</description>
      <url-pattern>/test/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <description>Basic Constraint</description>
      <role-name>Administrator</role-name>
    </auth-constraint>
  </security-constraint>
```

RESTful API resources constraints

- In EntityFacadeREST.java, use
- @DeclareRoles(“{Admin,Regular}”)
  - On the root resource to declare the roles that apply
- @RolesAllowed(“Admin”)
  - On the method for the path to be protected

Turn on Protection for Faces Pages

- Arrange for users to exist, with userid, password, group
  - File realm in glassfish: users added manually
  - Database table: users can be added dynamically

- Turn on Authentication with an annotation in a Java file (often a servlet file, but for our purposes we'll use ApplicationConfig as a convenient class name):

```
@BasicAuthenticationMechanismDefinition
@ApplicationScoped
@Named
public class ApplicationConfig {  }
```

- Specify which resources are protected, and which roles are allowed to access those resources (in web.xml)

Turn on protection for RESTful API resources

- Arrange for users to exist, with userid, password, group
  - File realm in glassfish: users added manually
  - Database table: users can be added dynamically

- Turn on Authentication with an annotation in ApplicationConfig.java extending Application

```
@BasicAuthenticationMechanismDefinition
@ApplicationScoped
@Named
public class ApplicationConfig extends Application{…
}
```

- Specify which resources are protected, and which roles are allowed to access those resources (`@DeclareRoles, @RolesAllowed` in `EntityFacadeRest.java`)

Turn on authentication in one file

- For RESTful API we use the authentication annotation in ApplicationConfig.Java:

```
public class ApplicationConfig extends Application {
```

- For JSF Pages, use use the authentication annotation in a servlet source file, OR an ApplicationConfig file
- Turning on authentication for both at the same time
  - Use the RESTful API ApplicationConfig (which extends Application)
    - web.xml to define the security constraints for Java Server Faces
    - Annotations in EntityFacadeREST for RESTful resources

Declarative Security for address-book

- **Declarative:** important programming paradigm where the programmer describes the solution, and “the system” figures out how to realize the solution
- Solution in our case is that the system will block users from **a set of URLs** in our app unless they are **authenticated** to have an **adequate role**

- declare JUST these things to add authentication to address-book:
  - What URLS of the app are protected, what Roles are required for those URLS (that’s it! – no implementation). Security Constraint
  - Where are passwords stored (DatabaseIdentityStore or file realm)
  - Where are usernames stored (DatabaseIdentityStore or file realm)
  - Where are groupnames stored (DatabaseIdentityStore or file realm)

Basic Auth Address-book

- Basic Authentication for address-book application using file realm

1. Turn on authentication

   - OLD way (in web.xml)

```
<login-config>
        <auth-method>BASIC</auth-method>
</login-config>
```

   - NEW way (annotation in a Java file, like `ApplicationConfig.java`)

```
@BasicAuthenticationMechanismDefinition
```

Basic Auth Address-book (cont'd)

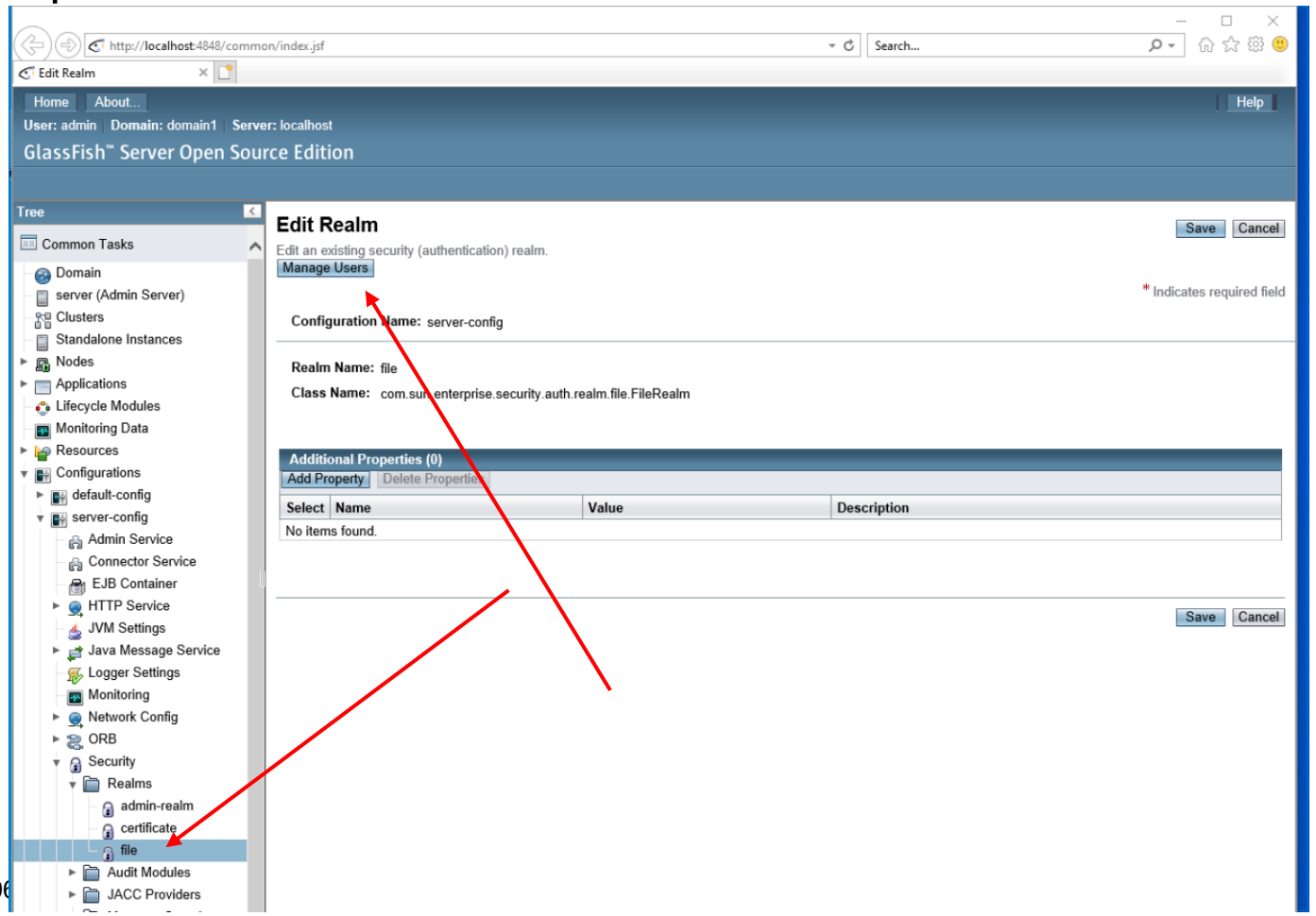
2. Specify Security Constraints in web.xml (NEW way is same as OLD way)

```
<security-constraint>
    <display-name>Constraint2</display-name>
    <web-resource-collection>
        <web-resource-name>Administration Area</web-resource-name>
        <description>Accessible to admin users only</description>
        <url-pattern>/dev/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description>Admin Area</description>
        <role-name>Devel</role-name>
        <role-name>Admin</role-name>
    </auth-constraint>
</security-constraint>
```

Basic Auth Address-book (cont'd)

3. Arrange for users to exist with usernames, passwords, and groups

One option is to use the file realm:



Basic Auth from database

- Basic Authentication for address-book application using a database table
- Use a @DatabaseIdentityStoreDefinition

```
@DatabaseIdentityStoreDefinition(
   dataSourceLookup = "${'java:comp/DefaultDataSource'}",
   callerQuery = "#{'select password from app.appuser where userid = ?'}",
   groupsQuery = "select groupname from app.appuser where userid = ?",
   hashAlgorithm = PasswordHash.class,
   priority = 10
)
```

- To use this @DatabaseIdentityStoreDefinition , we need a table called `appuser` with fields `userid`, `password`, `groupname`
- We can clone the address-book application into a new application that manages appusers (this is Lab4)
  - Clone the address-book application and in the process, change the Entity to AppUser with those needed fields

Code to create a password hash

- Notice that the Database Identity Store will use a PasswordHash class to compare the User's password attempt to the stored password entry for that user

- Here is code that will create password entries that can be checked with the default PasswordHash class

```
//set password entry
// initialize a PasswordHash object which will generate password hashes
Instance<? extends PasswordHash> instance = CDI.current().select(Pbkdf2PasswordHash.class);
PasswordHash passwordHash = instance.get();
passwordHash.initialize(new HashMap<String,String>());  // todo: are the defaults good enough?
// now we can generate a password entry for a given password
String passwordEntry = current.getPassword(); //the user has chosen a password
passwordEntry = passwordHash.generate(passwordEntry.toCharArray());
```

Password checking

- User creates an account with a new password, the PasswordHash implementation will
  1. generate a random salt value
  2. Add the salt value to the password
  3. Hash the salted password to get the hashed value
  4. Store the password entry as `algorithm:salt:hash`

`Later on…`

- User enters their claimed password to authenticate
  - Retrieve the `algorithm:salt:hash` for that user from the database
  - Add the salt to the claimed password
  - Hash the salted claimed password with the algorithm to get claimed_hash
  - Compare claimed_hash to hash and if they are the same, the claimed password is the same as the stored password

Summary

- To protect Faces pages
  - Annotations in some named application-scoped bean (the one used for RESTful API below, will work)
    - Turn on Authentication
    - Specify the DatabaseIdentityStoreDefinition
  - Security Constraints go in web.xml
    - Specify the resource as a URL pattern
    - Specify the roles allowed to access that resource
- To protect RESTful API resources
  - Annotations in `ApplicationConfig exends javax.ws.rs.core.Application`
    - Turn on Authentication
    - Specify the DatabaseIdentityStoreDefinition
  - Security Constraints go in EntityFacadeRest.java
    - Specify @RolesAllowed("…") as an annotation on each resource

Summary again in table form

	JSF Pages	RESTful API
Users, Groups	File realm or @DatabaseIdentityStoreDefinition	File realm or @DatabaseIdentityStoreDefinition
Role Mapping	Default Group->Role or web.xml (in this course we tend to use the default group->role mapping, so every group is a role)	Default Group->Role or web.xml (in this course we tend to use the default group->role mapping, so every group is a role)
Turn Authentication On	web.xml OR @BasicAuthenticationMechanismDefinition on a configuration class with @ApplicationScoped @Named (an application-scoped Named bean) OR @FormAuthenticationMechanismDefinition on a configuration class with @ApplicationScoped @Named (an application-scoped Named bean)	@BasicAuthenticationMechanismDefinition on a configuration class in the app that extends javax.ws.rs.core.Application
Protect Resources	web.xml security constraint with a URL pattern	@DeclareRoles (at the top) @RolesAllowed on method In a class with a @PATH annotation For us, that is EntityFacadeREST.java (for example, SpriteFacadeREST.java)