Access control subsystem

Contents

- Roles – advanced topics
- Anonymous access to the application UI
- Programmatic user management
- Additional security features
- Developing User-aware addons in Jmix
- External authentication services (KeyCloak)

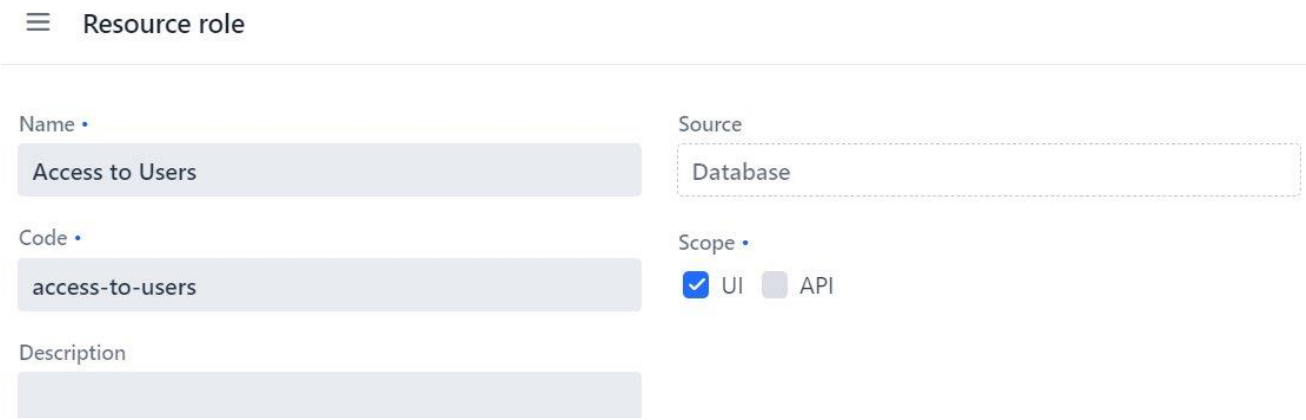
Roles – advanced topics

Role scopes

Specific policies

Role scopes

- **Scope** – type of the client
  - **UI** - Vaadin-based UI
  - **API** – Externally exposed API: REST API or GraphQL
- Role can have one or several scopes
- Scopes allow to define different set of roles to the user, depending on the type of client
- Role permissions are applied to the user only if type of the client used in the session is contained in the role scope




```
@ResourceRole(name = "AccessToUsersRole",
code = AccessToUsersRole.CODE, scope = "API")
public interface AccessToUsersRole {
    String CODE = "access-to-users-role";
}
```

Role scopes – motivation

- Vaadin-based UI
  - UI permissions - ability to hide view or menu item
  - Ability to disable or hide (programmatically) any view component (e.g. Button or DataGrid action)
  - Ability to show limited fraction of entity rows in any view
  - Client layer **is trusted**. Own web-page-to-server protocol, all checks are executed on server
    - Impossible to mock clicking a disabled button with a specially constructed HTTP request
- WebServices APIs
  - No concept of view
  - CRUD operations and services are exposed as is
  - Client (mobile app or web portal using the API) **is untrusted**. API communication protocol is open to 3rd-party for research
- As a result – *generally* **"API"** roles must have **more restrictions** than UI ones.
  - Many built-in framework roles have "UI" scope and cannot be used for API access

Role scopes - example

Requirement:

- Provide the ability to view user's in-app messages (sent or received)
- Don't allow to view other user's messages.

UI solution:

- Create a view dedicated for this feature. It displays only messages received or sent by the current user. The query is hard-coded into the view
- Assign a "UI" role:
  - Allow: Message (READ), all attributes
  - Allow: view & menu item "Messages"

Solution with external API:

- Assign a role with "API" scope:
  - Allow: Notification (READ), all attributes
- Assign a row-level role with "API" scope
  - JPQL (or Predicate) policy that hides all non-suitable messages from the user

```
<loader id="notificationsDl">
<query>
<![CDATA[select e from Notification e
        where e.recipient.id = :current_user_id]]>
</query>
</loader>
```

Role scopes - recommendations

- Use scope = UI to all roles developed for UI users.
- Carefully develop API roles for API users.
  - Limit what API users can do with row-level roles.
  - Create custom endpoints for business operations
    - E.g. it would be very hard to implement "change own password" securely with standard CRUD REST.

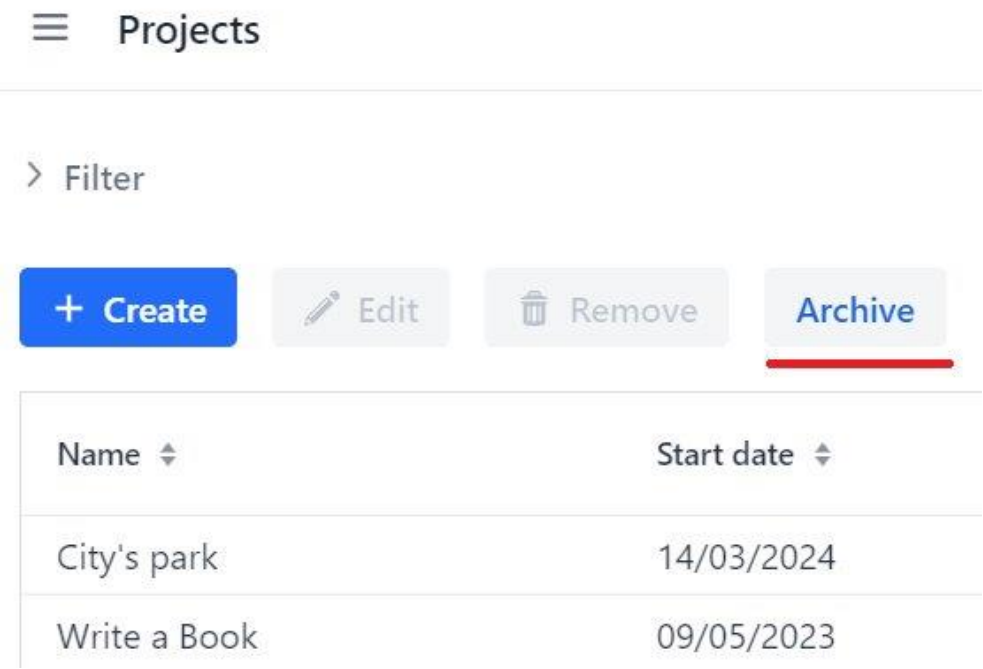
Specific policies

- Sub-element of the **resource** role.
- Define binary permissions on arbitrary named functionality that doesn't map well to CRUD
- Checked manually in the source code by the developer.
- Common misuse:
  - "How can I write hasRole("role") in Jmix?"
  - Use instead: hasSpecificPermission("specific-permission")
- Built-in specific policies:
  - "bulkeditor.edit.enabled" - *is user allowed to use Bulk Edit feature?*
  - "ui.loginToUi" - *is user allowed to login to UI?*

Specific policies - example

Action "Archive" in the Projects view - only for users with a permission.

Separate from Project entity CRUD permissions.




```
<dataGrid id="projectsDataGrid"
          width="100%"
          minHeight="20em"
          dataContainer="projectsDc"
          columnReorderingAllowed="true">
    <actions>
        <action id="create" type="list_create"/>
        <action id="edit" type="list_edit"/>
        <action id="remove" type="list_remove"/>
        <action id="archive"
                text="msg://projectsDataGrid.archive.text"
                type="list_itemTracking"/>
    </actions>
```

Specific policies - example

Define specific policy class

```
import io.jmix.core.accesscontext.SpecificOperationAccessContext;

public class ArchiveProjectContext extends SpecificOperationAccessContext {
    public static final String NAME = "pm.projects.archive";

    public ArchiveProjectContext() {
        super(NAME);
    }
}
```

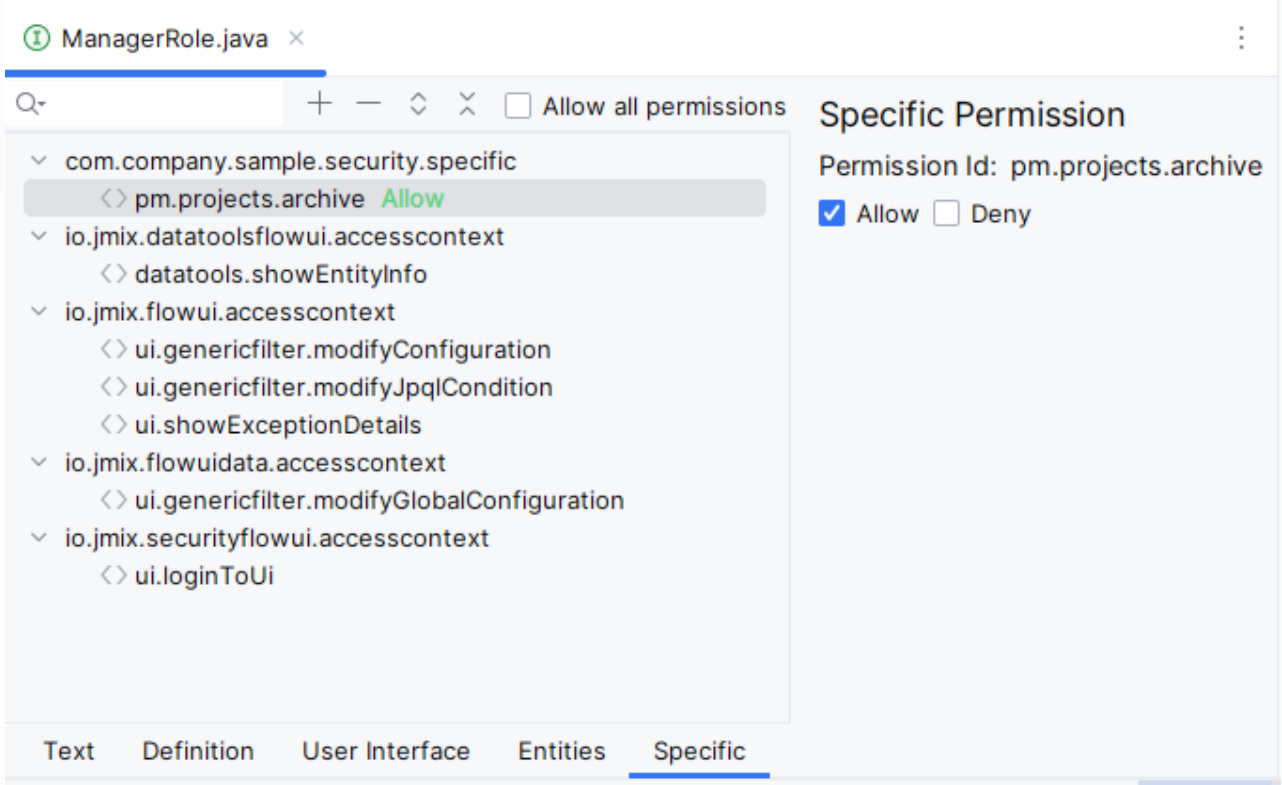
Use specific policy in the code

```
public class ProjectListView extends StandardListView<Project> {

    @Autowired
    private AccessManager accessManager;


    @Install(to = "projectsDataGrid.archive", subject = "enabledRule")
    private boolean projectsDataGridArchiveEnabledRule() {
        ArchiveProjectContext context = new ArchiveProjectContext();
        accessManager.applyRegisteredConstraints(context);
        return context.isPermitted();
    }
}
```

Specific policies - example

Add permission to the **resource** role

```
@ResourceRole(name = "ManagerRole", code = ManagerRole.CODE, scope = "UI")
public interface ManagerRole {
    String CODE = "manager-role";

    @SpecificPolicy(resources = ArchiveProjectContext.NAME)
    void specific();
}
```



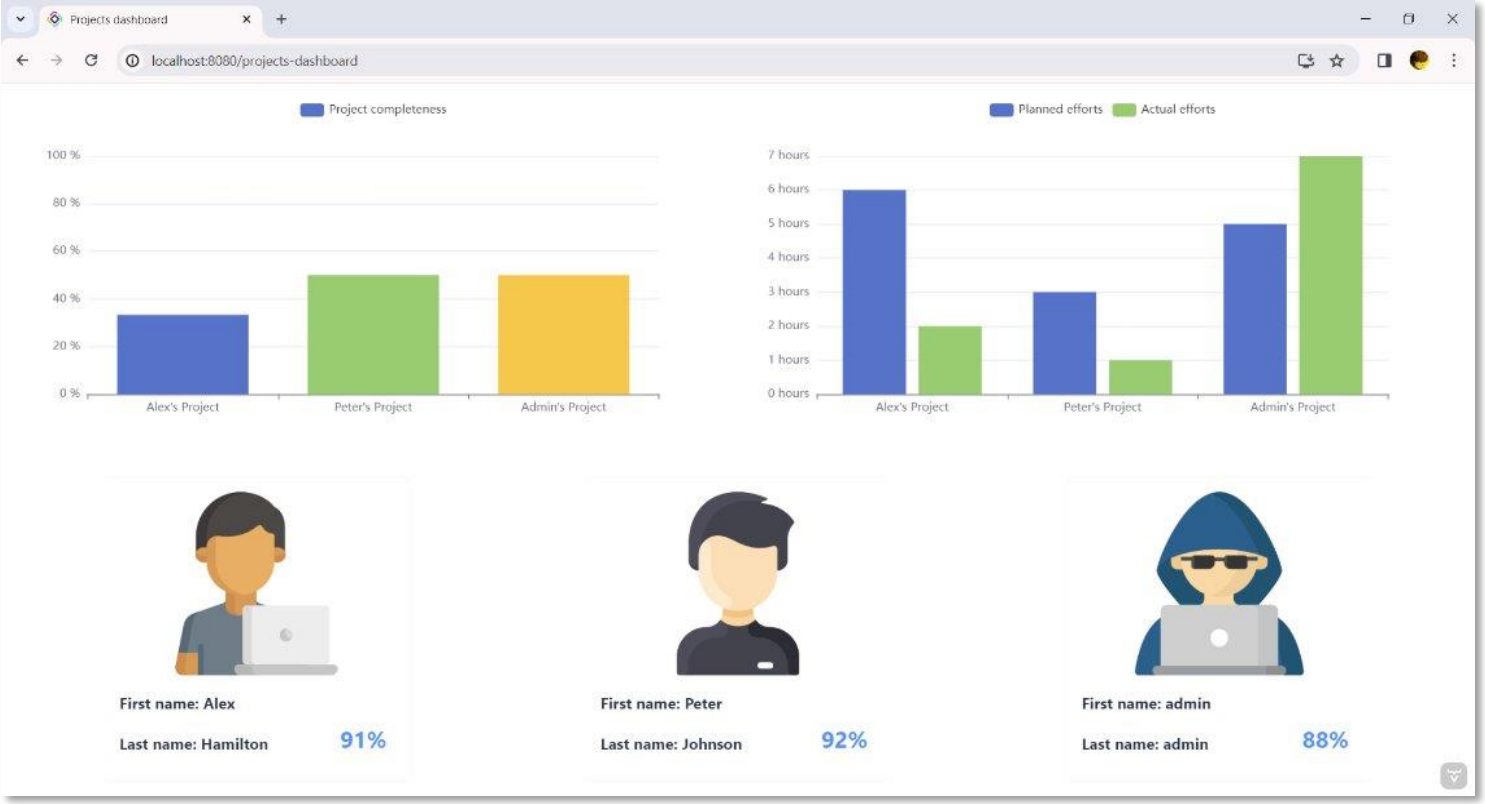
Anonymous access to application UI

Anonymous access to application UI

- **By default:**
  - User needs to log in before accessing any view.
  - Is automatically redirected to login view.
- *But what is we need to open some views for **everyone**?*
- Configure constraints for anonymous user:
  - Views, menu
  - Entities, attributes

Sample: public projects dashboard

- It displays graphs by project
- Can be accessed without logging in by using a link.
- + bonus: anonymous access to task list view.



Anonymous access to views – setting up #1

- Set up **anonymous resource role** (in code)
  - Enable all necessary views, menus, entities and attributes.
- Assign role(s) to **anonymous user**
  - Anonymous – special user in code
  - Method *DatabaseUserRepository#initAnonymousUser(User)*
  - Use *GrantedAuthoritiesBuilder* builder class

```
@Override
protected void initAnonymousUser(User anonymousUser) {
    Collection<GrantedAuthority> authorities = getGrantedAuthoritiesBuilder()
            .addResourceRole(AnonymousRole.CODE)
            .build();
    anonymousUser.setAuthorities(authorities);
}
```

Anonymous access to views – setting up #2

@AnonymousAllowed – can be used instead of View permissions from a resource role.

- Remove all View permissions from Resource Role
- Add annotation to views

```
@AnonymousAllowed
@Route(value = "tasks", layout = MainView.class)
@ViewController("Task_.list")
@ViewDescriptor("task-list-view.xml")
@LookupComponent("tasksDataGrid")
@DialogMode(width = "64em")
public class TaskListView extends StandardListView<Task> {
```

```
@AnonymousAllowed
@Route(value = "projects-dashboard")
@ViewController("ProjectsDashboardView")
@ViewDescriptor("projects-dashboard.xml")
public class ProjectsDashboardView extends StandardView {
```

http://localhost:8080/tasks (внутри MainView)

http://localhost:8080/projects-dashboard

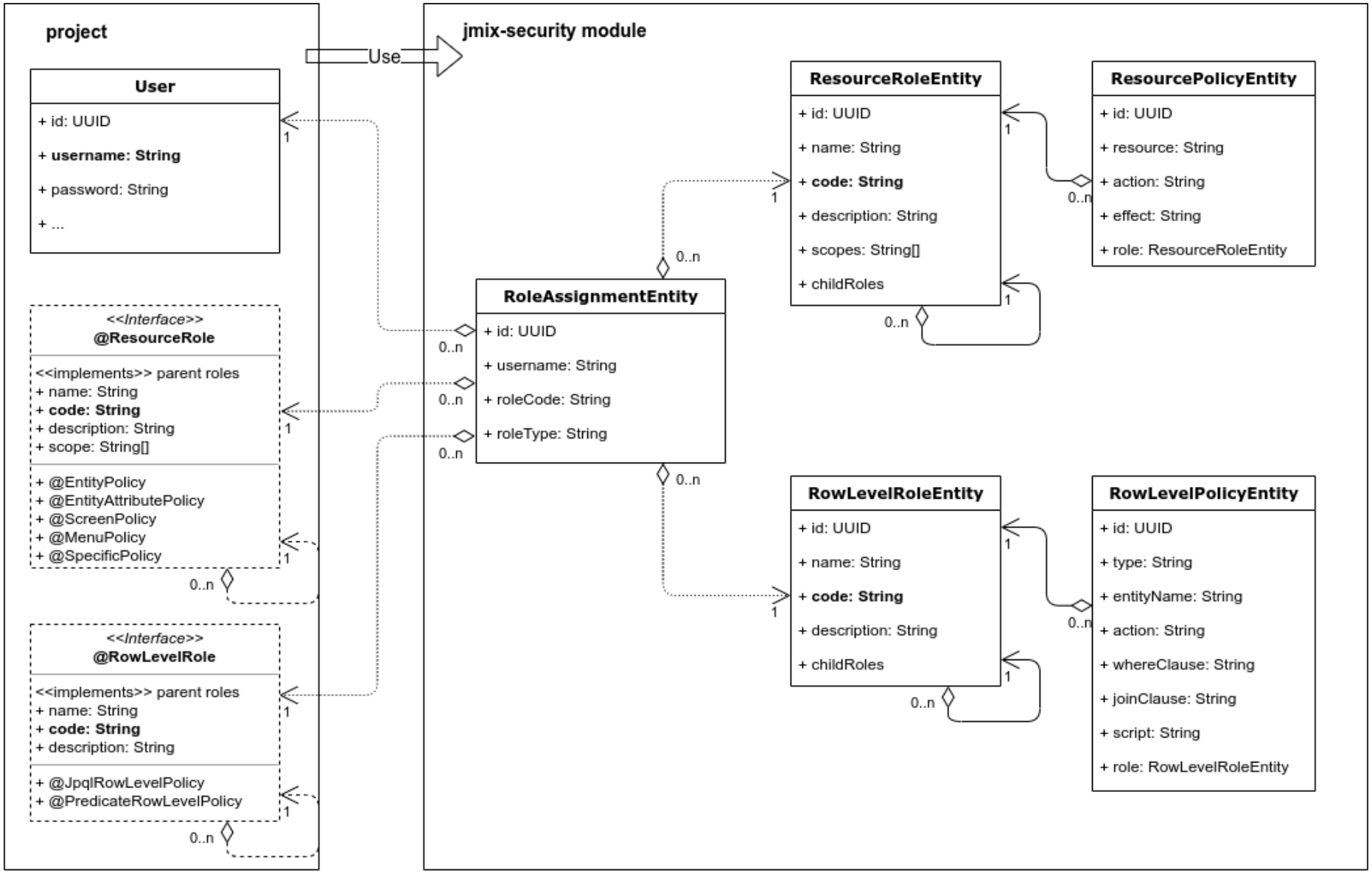
Programmatic user management

Programmatic user management

Contents:

- User and roles data model
- Creating users, assigning roles
- Password encryption overview
- Practical example: user registration and activation

User and roles – data model



User and roles – data model features

- Data model objects are spread between two modules: jmix-security and project
- Design-time roles exist only in source code as interfaces.
- Weak relations between RoleAssignmentEntity and User, RoleAssignmentEntity and roles.
- User is identified by unique, non-modifiable **username**
- Roles are identified by unique, non-modifiable **code**
- User entity requirements are very flexible:
  - Any base entity
  - Any primary key
  - Possible to not have User entity at all.

Creating users

No API provided by Jmix, just create entities manually. Example:

```
@Autowired
private UnconstrainedDataManager unconstrainedDataManager;

public User registerNewUser(String email, String firstName, String lastname) {
    User user = unconstrainedDataManager.create(User.class);
    user.setEmail(email);
    user.setUsername(email);
    user.setFirstName(firstName);
    user.setLastName(lastname);
    user.setActive(true);
    User savedUser = unconstrainedDataManager.save(user);
    return savedUser;
}
```

Assigning roles

No API provided by Jmix, just fill and save entities manually.

RoleType - one of **`io.jmix.security.role.assignment.RoleAssignmentRoleType`** constants

```
RoleAssignmentEntity ra1 = unconstrainedDataManager.create(RoleAssignmentEntity.class);
ra1.setRoleCode(CombinedManagerRole.CODE);
ra1.setRoleType(RoleAssignmentRoleType.RESOURCE);
ra1.setUsername(user.getUsername());

RoleAssignmentEntity ra2 = unconstrainedDataManager.create(RoleAssignmentEntity.class);
ra2.setRoleCode(RestrictedDocumentsRole.CODE);
ra2.setRoleType(RoleAssignmentRoleType.ROW_LEVEL);
ra2.setUsername(user.getUsername());

unconstrainedDataManager.save(ra1, ra2);
```

```
@ResourceRole(name = "CombinedManager", code = CombinedManagerRole.CODE)
public interface CombinedManagerRole extends ProjectManagementRole, DynamicAttributesRole, UiMinimalRole {
    String CODE = "combined-manager";
}
```

Password encryption

- Jmix uses:
  - ***spring-security-crypto*** module
  - org.springframework.security.crypto.password.PasswordEncoder
- The password is stored in the User#password persistent attribute.
- Passwords can be stored in raw or encrypted format.
- Multiple encryption methods can be used.
- Default encryption method is ***bcrypt***.

```
select PASSWORD from USER_ where PASSWORD is not null;
```

	PASSWORD
1	{bcrypt}$2a$10$35deOCuVwW0wYrLDBYbM/OQJm5zy1eVTwD4TDYfa40b1eNf1JdD9G
2	{noop}admin
3	{bcrypt}$2a$10$QDUY9bb7D4Gn3IOboi7o3.aigXQ2EIdgsJcG/cYIUq4l7kgiS/zUe
4	{noop}peter
5	{noop}alex
6	{noop}dev2
7	{bcrypt}$2a$10$XppyjixhupFQYbZ0cRn74u.OepgRFiwrO4slW25kr1WTxV3nknX.G

"Noop" encryption method

- Convenient encryption method for sample data during development.
- Don't use for production data!

```
<insert tableName="USER_" dbms="postgresql, mssql, hsqldb">
    <column name="ID" value="60885987-1b61-4247-94c7-dff348347f93"/>
    <column name="VERSION" value="1"/>
    <column name="USERNAME" value="admin"/>
    <column name="PASSWORD" value="{noop}admin"/>
    <column name="ACTIVE" valueBoolean="true"/>
</insert>
```

Using PasswordEncoder

- Encrypting entered password:

```
@Autowired
private PasswordEncoder passwordEncoder;

String rawPassword = passwordField.getValue();
String encryptedPassword = passwordEncoder.encode(rawPassword);
user.setPassword(encryptedPassword);
```

- Checking that entered password matches:

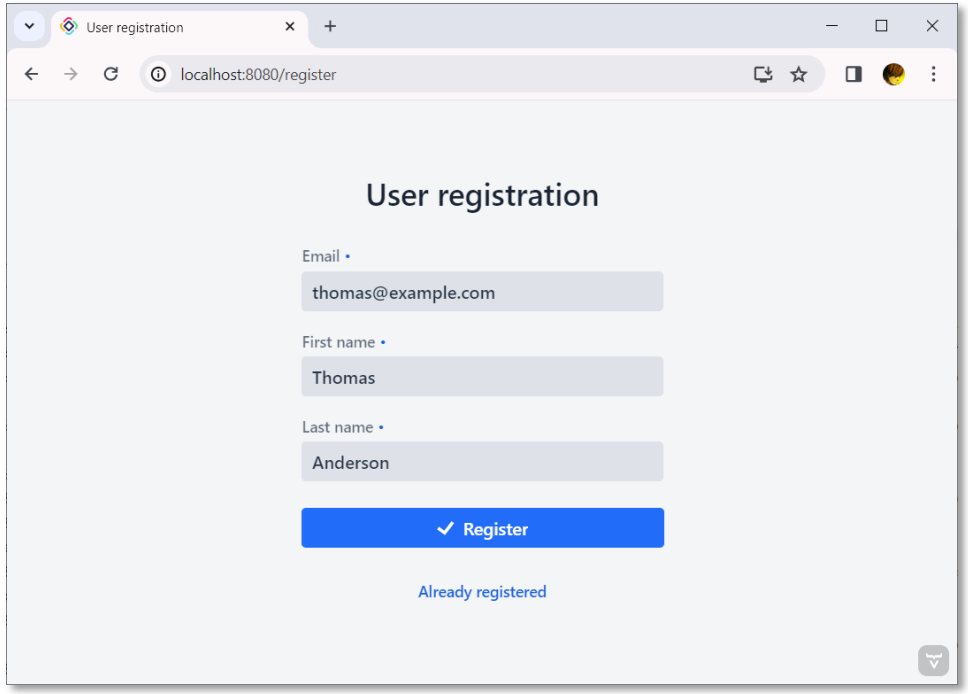
```
String enteredPassword = passwordField.getValue();
boolean passwordCorrect = passwordEncoder.matches(enteredPassword, user.getPassword());
```

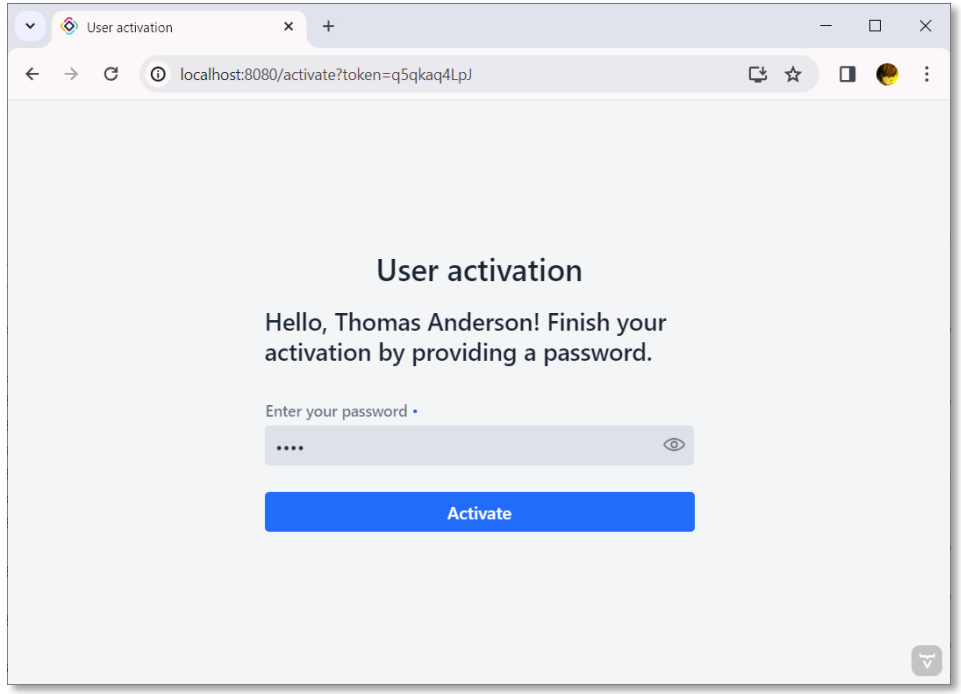
More information in the Spring docs:
https://docs.spring.io/spring-security/reference/features/integrations/cryptography.html#spring-security-crypto-passwordencoders

User registration example

- Link from login view to registration view
- Registration view is a publicly available
- Registration - by email
- User is created (in inactive state).
  - Activation link is sent to email.
  - Using http://nilhcem.com/FakeSMTP/ as mock SMTP server
- User clicks received activation link.
- User is directed to the activation view.
- User enters password.
  - The system activates user, assigns user roles and directs user to the main view.

User registration – implementation & demo





Additional security features

Additional security features

Included into Jmix framework and jmix-security module:

- Remember-me in UI
- System authentication
- Authentication events
- Brute-force protection
- User substitution

Remember-me in UI

- Works in (Vaadin) UI
- Allows user to persistently "remember" logged in state for a long time
- Integrated with spring-security mechanisms
- Tokens are stored in the PERSISTENT_LOGINS table
- Implementation -
  org.springframework.security.web.authentication.rememberme.JdbcTokenRepositoryImpl
- Token is returned with HTTP response as Cookie on successful login
- When session is lost, client will automatically log in if token isn't expired yet
  - jmix.core.rememberme.token-validity-seconds = 30 days by default

Output | PUBLIC.PUBLIC.PERSISTENT_LOGINS

1 row | Tx: Auto | DDL

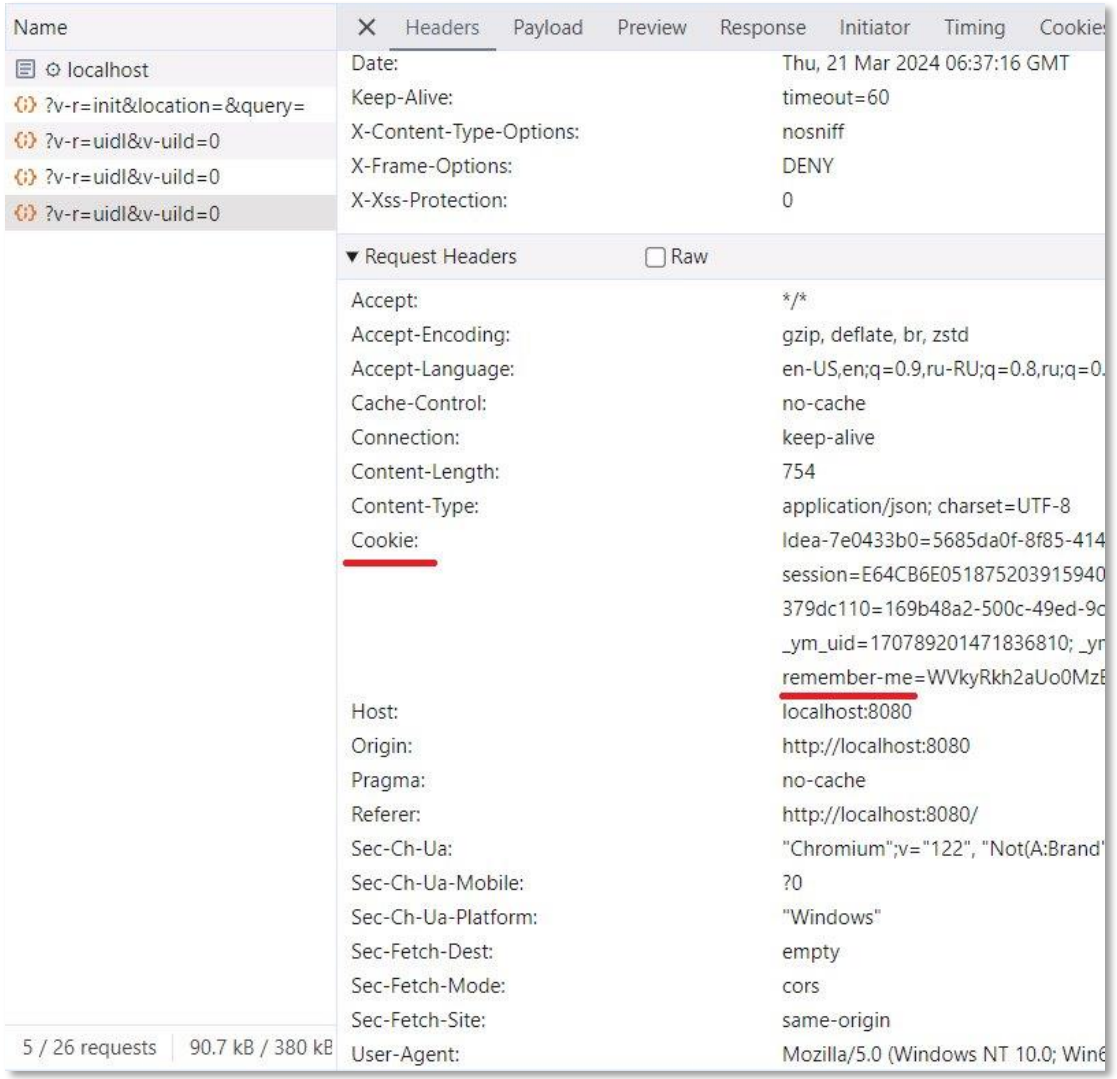
	SERIES	USERNAME	TOKEN	LAST_USED
1	lgD09LGQAl4KZ6R/z4iqeQ==	alex	NE0dbQLgPy+1yaa3SnC0AQ==	2021-11-01 19:32:09.423000

Deleting remember-me tokens

- When user changes password – tokens are deleted automatically
- Programmatically
  - `io.jmix.core.security.UserManager#resetRememberMe`

```
@Autowired
private UserManager userManager;

userManager.resetRememberMe(List.of(currentAuthentication.getUser()));
```

Remember-me in UI (demo)



System authentication

Motivation:

- Some Jmix mechanisms require authorization or authentication information:
  - DataManager data access checks.
  - Specific permissions checking with AccessManager.
  - ...
- There are internal code flows that don't have any authentication information:
  - Quartz schedulers,
  - JMX operations,
  - JMS topic listeners,
  - etc.

System authentication - API

- Bean ***io.jmix.core.security.SystemAuthenticator***
- *@io.jmix.core.security.**Authenticated*** annotation

Actions can be executed as:

- Any user (identified by username),
- Or built-in "System" user.

System user:

- Exists only in memory.
- Authorities are filled in *DatabaseUserRepository#initSystemUser()*
- By default - full access.

System authentication – usage examples

- Try-finally style, "registration-cleaner" user

```
@Autowired
private SystemAuthenticator systemAuthenticator;

@Override
public void execute(JobExecutionEvent context) {
    systemAuthenticator.begin("registration-cleaner");
    try {
        deleteOldNotActivatedUsers();
    } finally {
        systemAuthenticator.end();
    }
}
```

- Lambda style

```
@Override
public void execute(JobExecutionEvent context) {
    systemAuthenticator.withUser("registration-cleaner", () -> {
        deleteOldNotActivatedUsers();
        return null;
    });
}
```

- Method reference, "system" user

```
@Override
public void execute(JobExecutionEvent context) {
    systemAuthenticator.withSystem(this::deleteOldNotActivatedUsers);
}
```

System authentication example – Quartz job

Task:

- Store user creation date in *User#createdDate*
- Periodically delete User :
  - That have not finished activation
  - And createdDate >= '7 days ago'

Authentication events

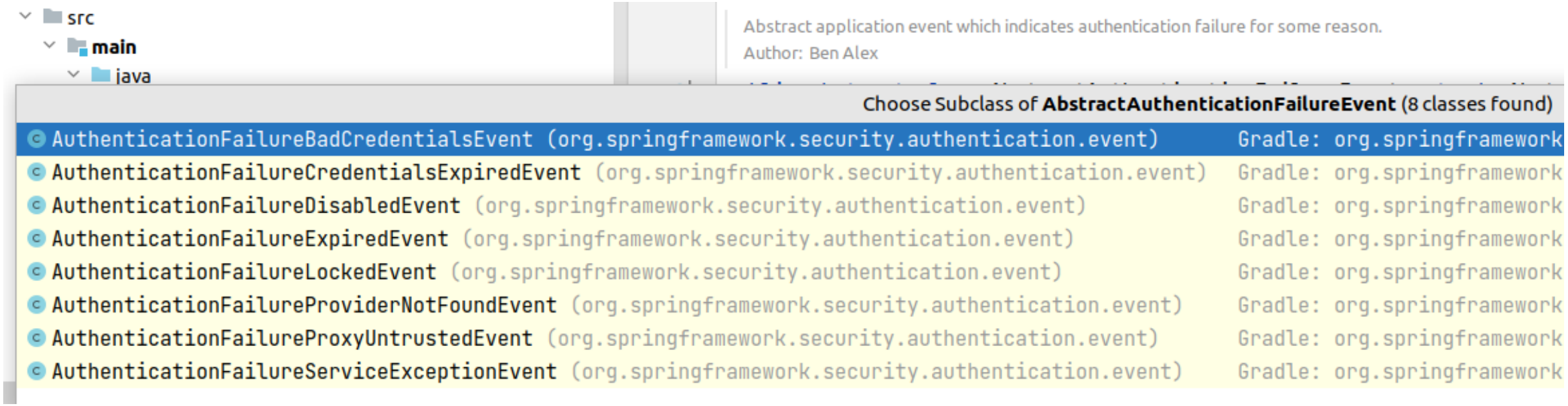
Authentication events serve two purposes:

- Be notified and be able to react on user login.
- Perform additional checks and stop the authentication process (by throwing an exception).

Application events published during authentication:

- ***AuthenticationSuccessEvent***
  - Just after authentication has succeeded. Jmix authentication isn't set yet.
- ***InteractiveAuthenticationSuccessEvent***
  - After fully completed login (authentication is set) in UI or REST.
- ***AbstractAuthenticationFailureEvent*** and its subclasses.
  - After various kinds of authentication failures (wrong password, user is disabled, etc.)
  - Base class, many more specific subclasses.

AbstractAuthenticationFailureEvent subclasses



Authentication events #2

Events published just before or after authentication credentials check procedure. Part of Jmix.

- ***PreAuthenticationCheckEvent*** (io.jmix.core.security.event)
  - Just before the authentication.
- ***PostAuthenticationCheckEvent*** (io.jmix.core.security.event)
  - Just after the authentication.

Can be used like "hooks" to pre-check / post-check the user. Throw exception to make login fail.

Example: checking User IP mask

GitHub: https://github.com/jmix-edu/sample-ip-mask

Using **PreAuthenticationCheckEvent** listener.

```
@EventListener
public void onPreAuthenticationCheck(PreAuthenticationCheckEvent event) {
    if (isProtectionEnabled()) {
        UserDetails userDetails = event.getUser();
        String mask = ((User) userDetails).getIpMask();
        if (!isValid(getIpAddress(), mask)) {
            throw new LockedException("You are not permitted to log in from this IP address");
        }
    }
}

private boolean isValid(String validatingIp, String ipMask) {
    return new IpMatcher(ipMask).match(validatingIp);
}
```

Authentication events – usage examples

Few examples of features how authentication events can be utilized:

- Checking user IP address before login.
- Remembering and pre-checking failed login attempts ("brute-force protection").
- Initializing additional session attributes.
- Recording a DB user session log.
- Counting number of active sessions (licensing limitations).
- Time access control for backoffice users.

Offtopic: additional UserDetails checks

- Interface org.springframework.security.core.userdetails.UserDetails provides many additional check methods.
- Not used by default Jmix User
- You can easily implement in your project.
- Example: implementing "User expiry date" feature with UserDetails#isAccountNonExpired()

> Indicates whether the user's account has expired. An expired account cannot be authenticated.
> Returns: `true` if the user's account is valid (ie non-expired), `false` if no longer valid (ie expired)

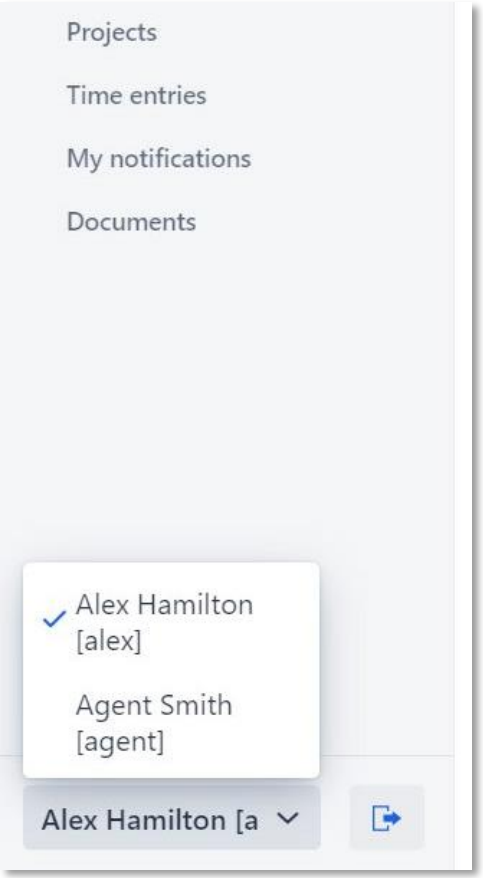
```
boolean isAccountNonExpired();
```

Brute-force protection

- Simple built-in protection against password brute force cracking.
- Listens for *AuthenticationFailureBadCredentialsEvent* (password not matched)
- Remembers multiple login attempts of pairs [login; IP address]
- Controlled with application properties:
  - jmix.security.bruteforceprotection.enabled - disabled by default
  - jmix.security.bruteforceprotection.maxLoginAttemptsNumber - 5 by default
  - jmix.security.bruteforceprotection.blockInterval - 60 seconds by default
- More info: https://docs.jmix.io/jmix/security/authentication.html#brute-force-protection
- Recommended to turn on for all production deployments (unless you use more sophisticated protection).

User substitution

- System admin can give a user an ability to ***substitute*** another user.
- The same session, but different set of permissions (resource and row-level roles).
- Management available at: Menu -> Users -> User substitutions.
- Substitution control in the corner of the view.
- Bean io.jmix.c.u.CurrentUserSubstitution to obtain substituted user in code.
- Docs: https://docs.jmix.io/jmix/whats-new/index.html#user-substitution



Accessing User
in Jmix framework and add-ons

User architecture - problem

- User entity class isn't in framework
- User entity is **optional**
- Framework cannot depend on project

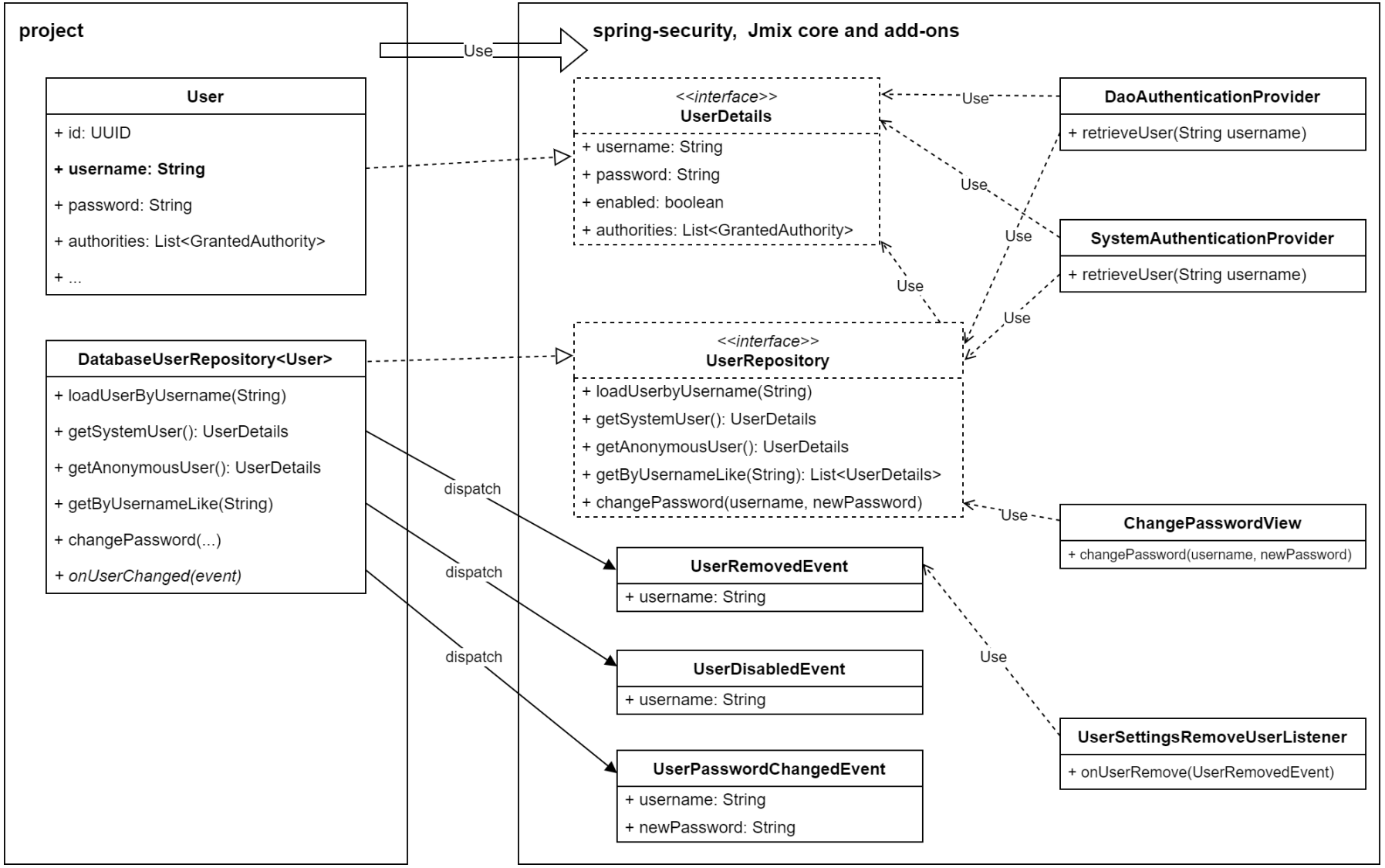
How add-ons can implement User-related operations?

- Load user by username (for login)
- Change user password
- Cleanup associated data after User removal

User architecture - solution

- "Inversion of control" - make both framework and project depend on common interface
  - UserDetails - common interface for User object.
  - UserRepository – common interface for User management logic.
  - User-related framework logic calls interface methods, doesn’t know implementation.
- Publication of User-related events
  - Project's bean publishes UserRemovedEvent, UserDisabledEvent, other events.
  - Interested add-ons listen to these events.
- Users are identified by unique, non-modifiable *username*.
  - Username is supported by all authentication services.

UserDetails & UserRepository architecture



UserRepository responsibilities

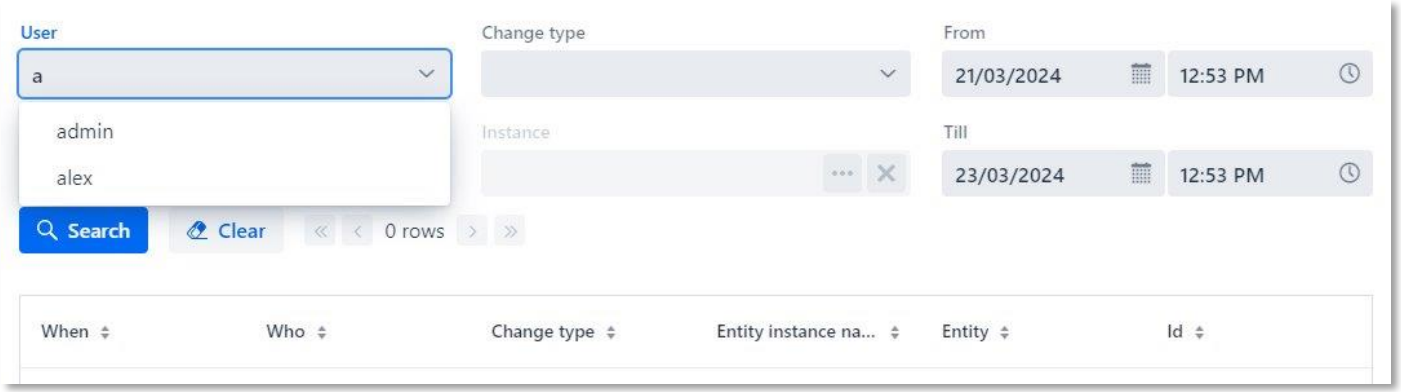
Contains all logic to init / load / modify users:

- Load user by username (for authentication).
- Load users list by search string (for UI controls).
- Change & save user password, reset passwords (for UI actions).
- Obtain instance of system and anonymous users.

Default implementation – JPA entity (User).

Logic can be overridden in the project (e.g. for LDAP integration – delegate searching users to LDAP server).

Entity Log - UserRepository usage example

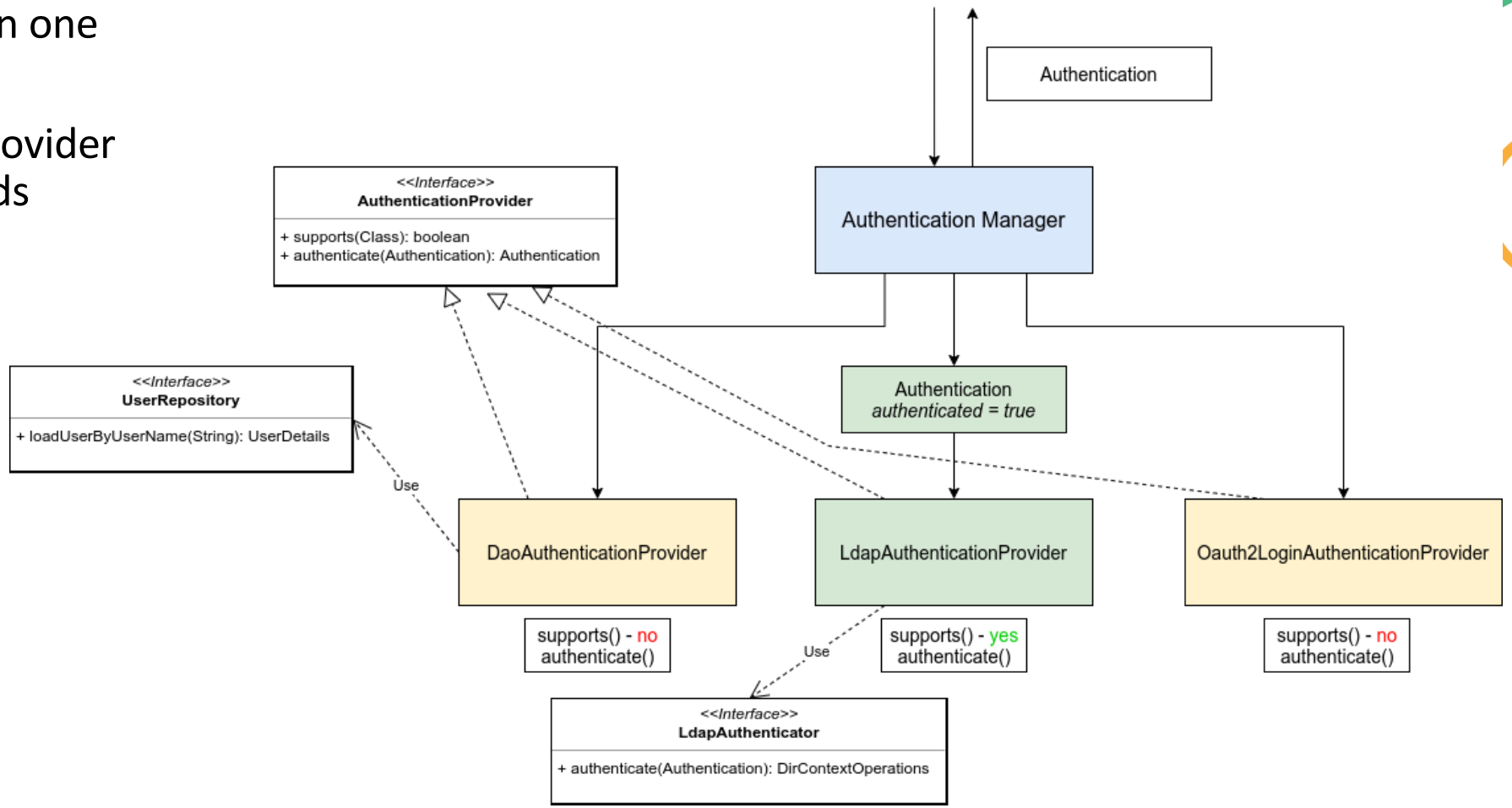



```
<comboBox id="userField"
          label="msg://user"
          autoOpen="true"/>
```

```
protected void onInit(View.InitEvent event) {
userField.setItemsFetchCallback(this::onUserFieldFetchCallback);
}

protected Stream<String> onUserFieldFetchCallback(Query<String, String> query) {
   return StringUtils.isNotBlank(enteredValue)
   ?
userRepository.getByUsernameLike(enteredValue).stream().map(UserDetails::getUsername)
      : Stream.empty();
}
```

Authentication providers in spring-security

- Several providers can exist in one application
- Authentication calls each provider until authentication succeeds
- UserRepository is used by DaoAuthenticationProvider



More details: https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#page-title

External authentication services

Single Sign-on with KeyCloak

External authentication - introduction

- By default in Jmix projects - local authentication:
  - Log in by username and password (UsernamePasswordAuthenticationToken).
  - Users are stored in relational database.
  - User management: programmatic, via Users UI view or via external API.
  - Role assignments – also in local DBMS.
- Benefits:
  - Fast project start.
  - Convenient development, no dependency on external services.
  - Storage of additional useful attributes in the User entity.
  - Can link other entities to User in the project data model.

External authentication - introduction

Not everyone is happy with local auth:

- Big organizations:
  - Have centralized user database in LDAP / AD / KeyCloak.
  - Wish to automatically deactivate user when the employee retires.
  - Spread of SSO solutions (KeyCloak) - one authorization server for many services.
  - Authority assignment - also convenient to perform in a centralized way.
  - Trends: infrastructure is moved to clouds, including users database (e.g. AWS Cognito).
- Public portals:
  - Inconvenient to enter private data and invent new password for every site.
  - Integration with social networks is widespread. Automatic log in and filling of user profile details.

External authentication in Jmix

Jmix isn't hard to integrate with external authentication services.

- Security subsystem is based on *spring-security* and compatible with its modules.
- No hard coupling to the User entity (abstractions: UserDetails, UserRepository).
- No hard coupling to RoleAssignmentEntity for authorization (determining user authorities).

Integration structure

Main tasks of Jmix integration module:

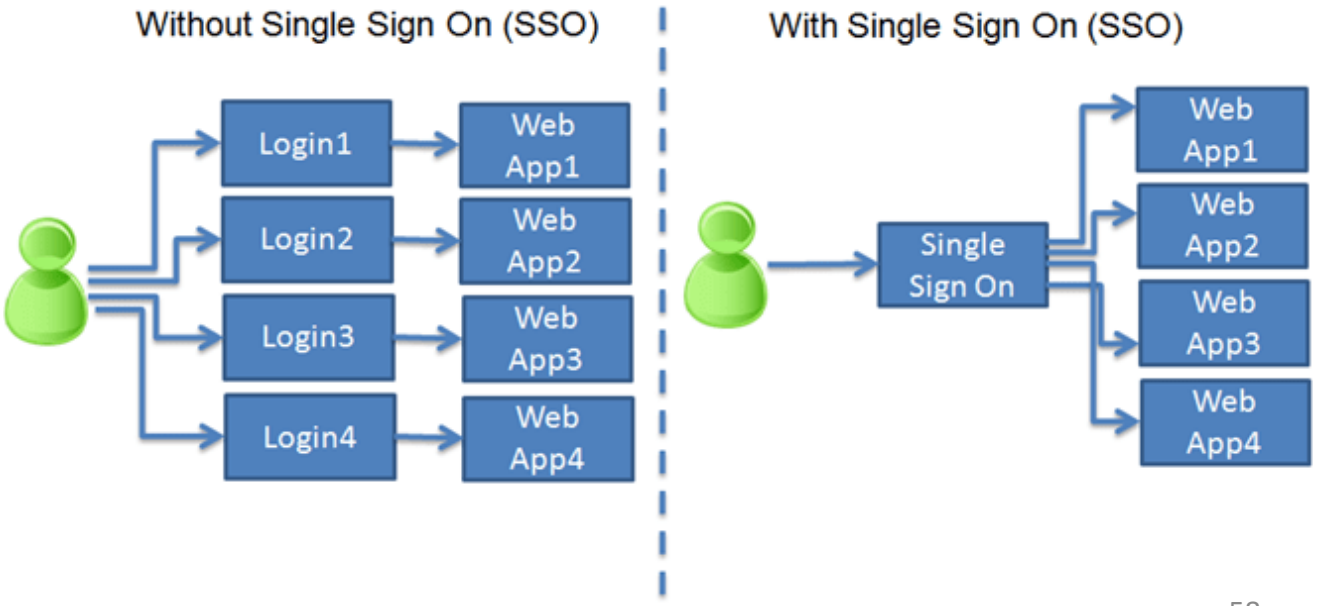
- Refreshing User details from external user, if it's necessary.
- Assigning User authorities based on external user attributes or group memberships
- Delegating `UserRepository` operations (user search) to external services.
- Performing logout in external SSO service when user logs out in Jmix application.

Single sign-on with KeyCloak

Single sign-on

Single sign-on (SSO) is:

Technology allowing users to switch from one application to another one, not related together, without needing to authenticate again.

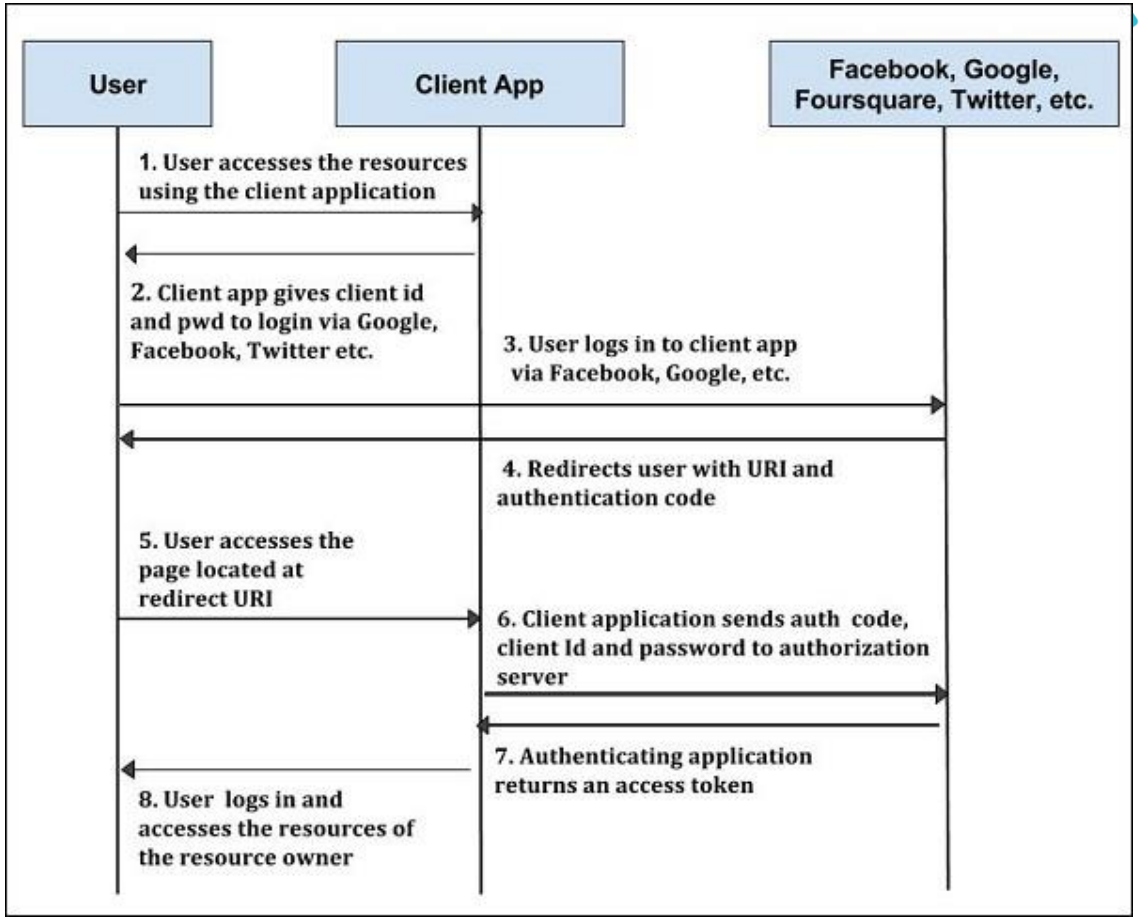


Pros and cons of SSO

- Advantages:
  - User convenience:
    - No need to spend time to enter passwords many times
    - Less chaos with various username / password combinations
  - Security:
    - Needs to implement high-quality password storage and authentication procedure only once, in the dedicated product
    - Easier to manage password policies (complexity, periodical change)
- Disadvantages:
  - Security: leaking a single SSO password --> disaster
  - Reliability: all applications depend on single SSO server

OAuth protocol

- Open authorization protocol
- Allows to grant one service rights to access user's resources stored on the other service
- Rescues from having to grant 3rd-party application credentials to other service
- Provides ability to grant limited set of permissions, not all of them at once.

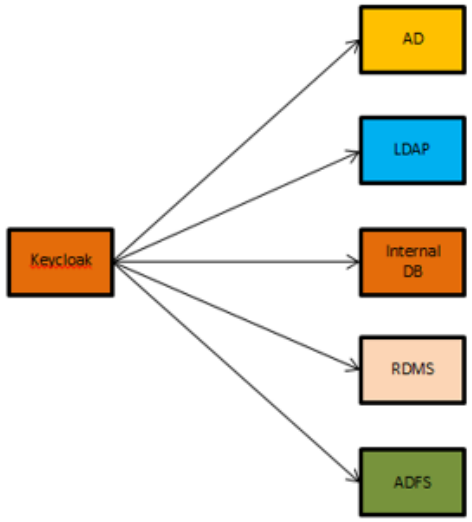


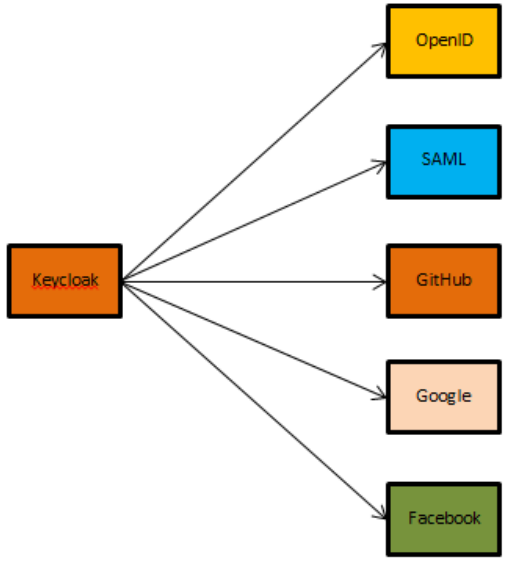
OpenID Connect (OIDC) protocol

- OIDC – thin layer on top of OAuth 2.0
- Adds username and profile information of the user who logged in.
- Facilitates implementing such scenarios when one login could be used in multiple applications (single sign-on).

What is Keycloak

- Open-source product, supported by RedHat
- Implements Single Sign-On with access control features.
- Supports lots of different things, including OpenID Connect.





KeyCloak Integration. Initial data

- Jmix project: https://github.com/jmix-edu/sample-sales
- Add-on OIDC https://docs.jmix.io/jmix/oidc/index.html
  - Protocol: OpenID Connect.
  - There is a User synchronization. User details are taken from KeyCloak User.
- KeyCloak server
  - Stores users and roles

(demo)

KeyCloak integration steps

- Set up KeyCloak:
  - Create and set up *client*, obtain client id / secret.
  - Create users
  - Create and assign roles
- Add OIDC add-on
- Create configuration beans
- Set up User synchronization. Integrate Jmix User with org.springframework.security.oauth2.core.oidc.user.OidcUser.

Any questions?