Module - Preparing a new CzechIdM module

Basic application skeleton you can create by archetype, please follow this tutorial: Archetype tutorial.

CzechIdM is a modular application built by Maven. Modules often have really similar structure consisting of:

- Java source code
- test suites
- Groovy scripts
- notification templates
- Activiti workflow definitions
- connector specific files (JDBC scripts etc.)
- configuration files and maven profiles
- database migration - Flyway configuration

In this tutorial we will describe the currently preferred way of setting up a CzechIdM module. Some basic information about developing the module can be found also in the Developer Guide - Quickstart.

Java source code

The Java code typically handles all the business logic and heavy lifting in the application. There are multiple areas of interest you will probably need while developing a module:

- adding configuration properties
- adding / removing event processors
- defining EAV attributes
- initialize application data
- overriding synchronization

Keep all Java sources under *src/main/java* directory and keep package structure as *eu/bcvsolutions/idm/#your_module_name#*.

Configuration properties

Setting default module configuration is crucial. The default configuration is loaded from the _application*.properties_ files, which will be discussed later. You can override the values in the properties by defining you own props in the application, which get stored in the database and have higher priority over the configuration files.

For accessing the properties, create a Spring bean extending the `Configurable` interface. See example implementation of module configuration, or EmailerConfiguration and its implementation DefaultEmailerConfiguration as an example. Creating a strongly typed configuration bean will give you great flexibility - you can use whatever types you need without being forced to parse the props from strings every time you need it.

It is advised to keep the configuration logic in the *config* Java package.

Event processors

Event processors are a powerful mechanism in customizing your CzechIdM deployment. All major entities, such as `IdmIdentity`, supports events. Use case example: sending notification to administrator after an identity has been created:

1. define an IdmIdentityDto processor with priority greater than 0 (after the identity is saved)
2. set the processor to catch only `IdentityEventType.CREATE` events
3. sent notification in the `process` method

That's it. Without the need to modify any of the existing code or the core module we have modified the behavior of the application.

This way you can alter entities processing, provisioning and many other cases.

Events also carry event properties. You can use such properties to cancel provisioning, for example, by the following code:

```
SysProvisioningOperation op = event.getContent();
op.setResultState(OperationState.CANCELED);
provisioningOperationService.delete(op);
return new DefaultEventResult<>(event, this, true);
```

Turning modules on and off is also possible in runtime by configuration, see the CzechIdM wiki for details.

It is advised to keep all processors in the *event* Java package and sub-packages.

EAV attributes

Extending entities in CzechIdM could be simply done by defining EAV attributes. These are referred by plain String keys in the code, which may become troublesome for larger modules, simply because magic values and restatements are generally a bad idea.

To keep EAV references type typo-safe, create a classic Java enum, which accepts two strings as constructor params. Example:

```
public interface EavConstants {

    String getEavCode();

    PersistentType getPersistentType();
}

public enum MyIdentityConstants implements EavConstants {

```

```
    MY_CUSTOM_SOMETHING("myCustomSomethingEav", PersistentType.TEXT),
    MY_ATTRIBUTE("myAttrEavCode", PersistentType.TEXT);

    // fields
    private final String eavCode;
    private final PersistentType persistentType;

    MyIdentityConstants(String eavCode, PersistentType persistentType) {
        this.eavCode = eavCode;
        this.persistentType = persistentType;
    }

    @Override
    public String getEavCode() {
        return eavCode;
    }

    @Override
    public PersistentType getPersistentType() {
        return persistentType;
    }
}
```

With such setup you can reference all your module-specific EAVs as `MyIdentityConstants.MY_ATTRIBUTE`. Another advantage of this approach is creating attribute definitions on application startup.

It is advised to keep the the EAV attributes definitions in the *constants* Java package.

Initialize module data

To ensure our module is easily runnable even on the first time it's deployed, we need to initialize module-specific data. That is done through Spring application event listener. This is highly recommended at least for creating EAV attribute definition (+ beneficial for tests, which start with empty database and we need to initialize the definitions somowhere). Following example show such bean, which creates EAV definitions for our `MyIdentityConstants` enum and create a role for notifications:

```
@Component
@DependsOn("initApplicationData")
public class InitMyModuleData implements
ApplicationListener<ContextRefreshedEvent> {

    @Autowired
    private SecurityService securityService;
    @Autowired
    private FormService formService;
    @Autowired
    private IdmRoleService roleService;
```

```
    @Autowired
    private MyConfiguration configuration;

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        init();
    }

    public void init() {
        // system auth. to get full permissions
        securityService.setSystemAuthentication();
        //
        try {
            for (EavConstants extAttr : MyIdentityConstants.values()) {
                createEavAttribute(extAttr, IdmIdentity.class);
            }
            //
            final String organizationsEventRecipientRole =
configuration.getNotificationRoleName();
            IdmRole orgRecipientRole =
roleService.getByName(organizationsEventRecipientRole);
            if (orgRecipientRole == null) {
                orgRecipientRole = new IdmRole();
                orgRecipientRole.setCanBeRequested(true);
                orgRecipientRole.setDescription("Role holders receive
notifications about our use case.");
                orgRecipientRole.setDisabled(false);
                orgRecipientRole.setName(organizationsEventRecipientRole);
                orgRecipientRole.setPriority(1);
                orgRecipientRole.setRoleType(RoleType.SYSTEM);
                roleService.save(orgRecipientRole);
            }
        } finally {
            SecurityContextHolder.clearContext();
        }
    }

    private void createEavAttribute(EavConstants extAttr, Class<? extends
FormableEntity> clazz) {
        IdmFormAttribute retrieved = formService.getAttribute(clazz,
extAttr.getEavCode());
        if (retrieved != null) {
            return;
        }
        //
        IdmFormAttribute attr = new IdmFormAttribute();
        attr.setCode(extAttr.getEavCode());
        attr.setName(extAttr.getEavCode());
        attr.setPlaceholder(extAttr.getEavCode());
        attr.setPersistentType(extAttr.getPersistentType());
        formService.saveAttribute(clazz, attr);
```

```
    }

}
```

Setting up test suites

Generally, all base classes for testing your module should be available out-of-the-box in the *czechidm-test-api* module, which you need to include as your module's dependency.

The only exception is for testing Activiti's workflows. For that reason, we need to create ActivitiRule bean first. Create following bean under the *test* sources module directory:

```
@ActiveProfiles("test")
@Configuration
public class MyModuleTestConfiguration {

    @Bean
    public ActivitiRule activitiRule(ProcessEngine processEngine) {
        return new ActivitiRule(processEngine);
    }
}
```

In version 7.3 it is also necessary to have dependency on *core-impl* module and create a base class for workflow testing. Following is a functioning example:

```
public abstract class AbstractMyModuleWorkflowIntegrationTest extends
AbstractWorkflowIntegrationTest {

    @Override
    public DefaultActivityBehaviorFactory getBehaviourFactory() {
        return new CustomActivityBehaviorFactory();
    }
}
```

With such setup you should be able to test Activiti workflows without any problems. For test example please refer to HistoryProcessAndTaskTest.

Without this setup, all your workflow tests will deliberately fail.

Groovy scripts

Groovy scripts are used primarily as transformations while connecting end-point systems.

It is advised to keep all your custom scripts the *src/main/resources/#package#/scripts* directory of

your module. Please do version the scripts, do not leave them only in the application! This is crucial for maintenance and upgrade reasons. XML is now a preferred script format, see IdmCoreGetFullName and IdmScript.xsd.

Notification templates

Keep notification templates in the *src/main/resources/#package#/templates* directory. For an example of core template, see IdmCoreChangeIdentityRole and IdmNotificationTemplate.xsd.

Activiti workflows

Activiti workflows are typically kept in the *src/main/resources/#package#/workflows* directory. Workflows are deployed and versioned automatically.

To test workflows, please see the `Setting up test suites` chapter.

Connector specific files

While connecting endpoint systems through scripted connectors (mostly), we have to either deploy these scripts with CzechIdM or we just simple need to version these scripts. The preferred way is to keep all connector scripts in the *src/main/resources/#package#/scripts* directory and sub-directories, however use whatever works best for you.

Please do version the scripts, do not leave them only in the application or on the server file system! This is crucial for maintenance and upgrade reasons.

For example of JDBC script development see the usage of Scripted SQL connector.

Database migration - Flyway configuration

Database migration (done by Flyway) is an important part of every db-backed application (module here) maintenance. Each module has its own Flyway configuration. To setup yours, follow the guide in wiki.