Quickstart - backend

Backend module is a separate jar file, which is loaded within CzechIdM. For the jar to be recognized as CzechIdM module, there needs to be a class implementing `ModuleDescriptor` interface (or extending `PropertyModuleDescriptor` which already implements some boring parts). The most trivial module needs to contain only:

```
@Component
@PropertySource("classpath:module-" + ExampleModuleDescriptor.MODULE_ID +
".properties")
@ConfigurationProperties(prefix = "module." +
ExampleModuleDescriptor.MODULE_ID + ".build", ignoreUnknownFields = true,
ignoreInvalidFields = true)
public class ExampleModuleDescriptor extends PropertyModuleDescriptor {

    public static final String MODULE_ID = "example";

    @Override
    public String getId() {
        return MODULE_ID;
    }
}
```

The module can also provide additional permission types or notification types - see the `AbstractModuleDescriptor` for particular methods. Don't forget to annotate the class with `@Component` to make it discoverable by Spring.

The module properties may be written to a new configuration file specific for the module. Create file **module-example.properties** where **example** is the same as `MODULE_ID` and put it in **src/main/resources** of the module. Set `@PropertySource` to the name of the file and Spring will load the properties file:

```
# module properties - not editable through configuration service (idm.pub.
prefix is not used)
# mapping pom.xml properties by default
# add custom properties if needed
#
# module version
module.example.build.version=@project.version@
# build number
module.example.build.buildNumber=@buildNumber@
module.example.build.buildTimestamp=@timestamp@
# module vendor
module.example.build.vendor=@project.organization.name@
module.example.build.vendorUrl=@project.organization.url@
module.example.build.vendorEmail=info@bcvsolutions.eu
# module description
module.example.build.name=@project.name@
module.example.build.description=@project.description@
#
```

```
## Example configuration properties - configurable from FE.
# test public property - public properties are available without login
idm.pub.example.public=public property value
# test private property - properties are available by permission
idm.sec.example.test.private=private property value
# confidential properties are stored in confidential storage, if key
contains reserved word ("password", "token", "secret")
# confidential properties are not send to FE
idm.sec.example.test.confidential.token=secret
```

remarks:

Even though CzechIdM looks like modular application, it is in fact monolithic. All the modules are always loaded during application startup. To switch off certain functionality of a module when the module is disabled, you need to annotate it with `@Enabled([MODULE_ID])`. This also allows to perform certain checks even when the module is disabled - like maintaining referential integrity.

Services (Beans)

The business logic is implemented inside services. A service should always consist of interface with required methods and a class implementing this interface annotated with `@Service`. Lifetime and dependencies of the services are managed by Spring. This allows us to replace implementation of a service with different one without writing additional code. Here is an example service:

```
/**
 * Example business logic - interface
 */
public interface ExampleService {

    /**
     * Returns given message.
     *
     * @param message
     * @return
     */
    Pong ping(String message);
}

/**
 * Example business logic - implementation
 */
@Service("exampleService")
public class DefaultExampleService implements ExampleService {

    private static final org.slf4j.Logger LOG =
org.slf4j.LoggerFactory.getLogger(DefaultExampleService.class);

    @Override
    public Pong ping(String message) {
```

```
        LOG.info("Ping [{}]", message);
        //
        return new Pong(message);
    }

}
```

This is example of service. Now we can have a basic service implementation and if some project want use different implementation, we can write another service where we override the default functionality without changing a line of code in the caller.

Event Processors

Important aspect of CzechIdM are events. As the name says, event is fired after a certain action occurs. Objects, which can process to the event and react to it. Read more about entity event processors and see the example.

Entities

Base entities used in CzechIdm are IdmIdentity and IdmRole.

Configuration

The module can have its specific configuration properties. Property names should follow conventions described in Application configuration.

First step is to create a new interface, which will extend the class **Configurable**. Add default methods and access methods for your properties:

```
/**
 * Example configuration - interface
 */
public interface ExampleConfiguration extends Configurable {

    static final String PROPERTY_PRIVATE =
ConfigurationService.IDM_PRIVATE_PROPERTY_PREFIX + "example.test.private";
    static final String PROPERTY_CONFIDENTIAL =
ConfigurationService.IDM_PRIVATE_PROPERTY_PREFIX +
"example.test.confidential.token";

    @Override
    default String getConfigurableType() {
        return "test";
    }

    @Override
    default List<String> getPropertyNames() {
```

```
        List<String> properties = new ArrayList<>(); // we are not using superclass properties - enable and order does not make a sense here
        properties.add(getPropertyName(PROPERTY_PRIVATE));
        properties.add(getPropertyName(PROPERTY_CONFIDENTIAL));
        return properties;
    }

    /**
     * Read private value from module-example.properties
     *
     * @return
     */
    String getPrivateValue();

    /**
     * Read confidential value from module-example.properties
     *
     * @return
     */
    String getConfidentialValue();
}
```

Create default implementation of your configuration interface by extending `AbstractConfiguration`. The module properties may be written to a new configuration file specific for the module. Create file **module-example.properties** where **example** is the same as `MODULE_ID` and put it in src/main/resources of the module. Set @PropertySource to the name of the file and Spring will load the properties file - if module descriptor extends `PropertyModuleDescriptor` itself, then is no need another `@PropertySource`.

```
/**
 * Example configuration - implementation
 */
@Component("exampleConfiguration")
public class DefaultExampleConfiguration
        extends AbstractConfiguration
        implements ExampleConfiguration {

    @Override
    public String getPrivateValue() {
        return getConfigurationService().getValue(PROPERTY_PRIVATE);
    }

    @Override
    public String getConfidentialValue() {
        return getConfigurationService().getValue(PROPERTY_CONFIDENTIAL);
    }
}
```

But it isn't required to create new properties file, if the properties have default value (see the class `IdentityConfiguration`), if they will be specified by external configuration or if you expect that

administrators will set them in GUI.

Configuration usage in service:

```
@Service("exampleService")
public class DefaultExampleService implements ExampleService {

    private final ExampleConfiguration exampleConfiguration;

    @Autowired
    public DefaultExampleService(ExampleConfiguration exampleConfiguration)
{
        Assert.notNull(exampleConfiguration, "Configuration is required!");
        //
        this.exampleConfiguration = exampleConfiguration;
    }

    @Override
    public String getPrivateValue() {
        return exampleConfiguration.getPrivateValue();
    }
        ...
}
```

Notifications

Each module can send notifications to CzechIdM identities and register used notification templates in module descriptor.

Register notification topic with default setting in module descriptor:

```
@Component
@PropertySource("classpath:module-" + ExampleModuleDescriptor.MODULE_ID +
".properties")
@ConfigurationProperties(prefix = "module." +
ExampleModuleDescriptor.MODULE_ID + ".build", ignoreUnknownFields = true,
ignoreInvalidFields = true)
public class ExampleModuleDescriptor extends PropertyModuleDescriptor {

    public static final String MODULE_ID = "example";
    public static final String TOPIC_EXAMPLE = String.format("%s:example",
MODULE_ID);

    @Override
    public String getId() {
        return MODULE_ID;
    }

    @Override
    public List<NotificationConfigurationDto>
```

```
getDefaultNotificationConfigurations() {
        List<NotificationConfigurationDto> configs = new ArrayList<>();
        configs.add(new NotificationConfigurationDto(
                TOPIC_EXAMPLE,
                null,
                IdmWebsocketLog.NOTIFICATION_TYPE,
                "Example notification",
                null));
        return configs;
    }
}
```

Websocket sender is configured for **example:example** topic, this can be changed in GUI. Notification can be send for example from service layer:

```
@Service("exampleService")
public class DefaultExampleService implements ExampleService {

    private static final org.slf4j.Logger LOG =
org.slf4j.LoggerFactory.getLogger(DefaultExampleService.class);
    private final ExampleConfiguration exampleConfiguration;
    private final NotificationManager notificationManager;

    @Autowired
    public DefaultExampleService(
            ExampleConfiguration exampleConfiguration,
            NotificationManager notificationManager) {
        Assert.notNull(exampleConfiguration, "Configuration is required!");
        Assert.notNull(notificationManager, "Notification manager is
required!");
        //
        this.exampleConfiguration = exampleConfiguration;
        this.notificationManager = notificationManager;
    }
...
    @Override
    public void sendNotification(String message) {
        notificationManager.send(
                ExampleModuleDescriptor.TOPIC_EXAMPLE,
                new IdmMessageDto.Builder()
                    .setLevel(NotificationLevel.SUCCESS)
                    .setMessage(message)
                    .build());
    }
...
}
```

REST Entry Points

All backend communication with outer world goes through HTTP REST services. The entry points are simple Java objects (called Controllers) configured only with annotations. You work with them like with normal objects and all the heavy plumbing is done by Spring. Here is a example Controller:

```
/**
 * Example controller
 */
@RestController
@Enabled(ExampleModuleDescriptor.MODULE_ID)
@RequestMapping(value = BaseController.BASE_PATH + "/examples")
@Api(value = "Examples", description = "Example operations", tags = {
"Examples" })
public class ExampleController {

    @Autowired private ExampleService service;

    @ResponseBody
    @RequestMapping(method = RequestMethod.GET, path = "/ping")
    @ApiOperation(
            value = "Ping - Pong operation",
            notes= "Returns message with additional informations",
            nickname = "ping",
            tags={ "Examples" },
            response = Pong.class,
            authorizations = {
                @Authorization(SwaggerConfig.AUTHENTICATION_BASIC),
                @Authorization(SwaggerConfig.AUTHENTICATION_CIDMST)
            })
    public ResponseEntity<Pong> ping(
            @ApiParam(value = "In / out message", example = "hello",
defaultValue = "hello")
            @RequestParam(required = false, defaultValue = "hello") String
message
            ) {
        return new ResponseEntity<>(service.ping(message), HttpStatus.OK);
    }

    @ResponseBody
    @RequestMapping(method = RequestMethod.GET, path = "/private-value")
    @ApiOperation(
            value = "Read private value",
            notes= "Returns configuration property - private value.",
            nickname = "getPrivateValue",
            tags={ "Examples" },
            authorizations = {
                @Authorization(SwaggerConfig.AUTHENTICATION_BASIC),
                @Authorization(SwaggerConfig.AUTHENTICATION_CIDMST)
            })
    public String getPrivateValue() {
        return service.getPrivateValue();
    }
```

```
    @ResponseBody
    @ResponseStatus(code = HttpStatus.NO_CONTENT)
    @RequestMapping(method = RequestMethod.GET, path = "/notification")
    @ApiOperation(
            value = "Send notification",
            notes= "Sending given message to currently logged identity
(example topic is used).",
            nickname = "sendNotification",
            tags={ "Examples" },
            authorizations = {
                @Authorization(SwaggerConfig.AUTHENTICATION_BASIC),
                @Authorization(SwaggerConfig.AUTHENTICATION_CIDMST)
            })
    public void sendNotification(
            @ApiParam(value = "Notification message", example = "hello",
defaultValue = "hello")
            @RequestParam(required = false, defaultValue = "hello") String
message) {
        service.sendNotification(message);
    }
}
```

The path where the controller is listening is configured via the class' @RequestMapping. In this case it is "/api/v1/examples". The base "/api/v1/" part is invariable. All paths with "public" after the base path are accessible for unauthenticated requests. Each method, which should be available through REST needs to be annotated with @RequestMapping. There you specify the correct HTTP method (GET, POST, ...) or a subpath if needed(which is concatenated with the path of the class).

As we can see, the method `ping` receives String parameter and returns `ResponseEntity<Pong>`. Those are mapped to HTTP bodies. To tell spring to map our objects to a body, we need to put `@RequestBody` before the parameter declaration and `@ResponseBody` before the method. The response needs to be wrapped inside a ResponseEntity to be able to specify HTTP status code and headers if needed or `@ResponseStatus` can be used. There are many more things you can configure with annotations in controllers, see the documentation.

Swagger annotations can be added for documentation reasons.

Errors

When some error occurs, then `ResultCodeException` should be thrown with appropriate `ResultCode`. Convention for codes with http status:

- 2xx - success messages (e.g. 202 - Accepted for asynchronous processing)
- 4xx - client errors (e.g. validations, bad values)
- 5xx - server errors ("our" IdM errors)

Example - result code definition

```
public enum ExampleResultCode implements ResultCode {

    EXAMPLE_CLIENT_ERROR(HttpStatus.BAD_REQUEST, "Example client error, bad value given [%s]"),
    EXAMPLE_SERVER_ERROR(HttpStatus.INTERNAL_SERVER_ERROR, "Example server error with parameter [%s]");

    private final HttpStatus status;
    private final String message;

    private ExampleResultCode(HttpStatus status, String message) {
        this.message = message;
        this.status = status;
    }

    public String getCode() {
        return this.name();
    }

    public String getModule() {
        return "example";
    }

    public HttpStatus getStatus() {
        return status;
    }

    public String getMessage() {
        return message;
    }
}
```

Example - send error

```
public void serverError(
        @ApiParam(value = "Error parameter", example = "parameter", defaultValue = "value")
        @RequestParam(required = false, defaultValue = "parameter") String parameter) {
    // lookout - ImmutableMap parameter values cannot be {@code null}
    throw new ResultCodeException(ExampleResultCode.EXAMPLE_SERVER_ERROR, ImmutableMap.of("parameter", String.valueOf(parameter)));
}
```

In service layer `ResultCode` with `ResultCodeException` can be used even with some success code - e.g. if some operation will be processed asynchronously, then `AcceptedException` can be used, if underlying service don't provide concrete api, e.g. override save method only and caller

doesn't know, if method will be synchronous or asynchronous. On controller layer `ResultModels` can be send as response without exception:

```
@ResponseBody
@RequestMapping(value = BaseController.BASE_PATH + "/public/result/info",
method = RequestMethod.PUT)
public ResultModels resultInfo() {
  ResultModels resultModels = new ResultModels();
  resultModels.addInfo(new DefaultResultModel(CoreResultCode.ACCEPTED));
  return resultModels;
}
```

Module data initialization

You may need to create some objects or perform some initialization tasks. For this, it is best to create initialization class beside the module descriptor:

```
/**
 * Initialize example module
 */
@Component
@DependsOn("initApplicationData")
public class ExampleModuleInitializer implements
ApplicationListener<ContextRefreshedEvent> {

    private static final org.slf4j.Logger LOG =
org.slf4j.LoggerFactory.getLogger(ExampleModuleInitializer.class);

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        LOG.info("Module [{}] initialization",
ExampleModuleDescriptor.MODULE_ID);
    }
}
```

This uses Spring's ApplicationListener which listens to the event when the application is initialized. It depends on initApplicationData which is initializer of the core module to ensure correct order.

This initializer is run everytime the application is started. This means also if the module is disabled!! You should ensure, that you don't create the data multiple times either by checking if they already exist or by setting a configuration property indicating the initialization already took place.

Tests

All features need to be tested. Unit tests are preferred way, but sometimes integration or rest test is easier to create.

Unit test for `ExampleService#ping` method:

```
/**
 * Example service - unit tests
 */
public class DefaultExampleServiceUnitTest extends AbstractUnitTest {

    @Mock
    private ExampleConfiguration exampleConfiguration;
    @Mock
    private NotificationManager notificationManager;
    @InjectMocks
    private DefaultExampleService service;

    @Test
    public void testPingWithMessage() {
        String message = "test";
        Pong pong = service.ping(message);
        Assert.assertNotNull(pong);
        Assert.assertEquals(message, pong.getMessage());
    }
}
```

Integration test for `ExampleService#getPrivateValue` method:

```
/**
 * Example service - integration tests
 */
public class DefaultExampleServiceIntegrationTest extends
AbstractIntegrationTest {

    @Autowired private ConfigurationService configurationService;
    @Autowired private ApplicationContext context;
    //
    private ExampleService service;

    @Before
    public void init() {
        service =
context.getAutowireCapableBeanFactory().createBean(DefaultExampleService.cla
ss);
    }

    @Test
    public void testPrivateValue() {
        String value = "valueOne";
        configurationService.setValue(ExampleConfiguration.PROPERTY_PRIVATE,
value);
        //
        Assert.assertEquals(value, service.getPrivateValue());
    }
```

```
}
```

Rest test for `ExampleController#ping` method:

```
/**
 * Example service - rest tests
 */
public class ExampleControllerRestTest extends AbstractRestTest {

    @Autowired private IdmIdentityService identityService;
    @Autowired private ModuleService moduleService;

    private Authentication getAuthentication() {
        return new
IdmJwtAuthentication(identityService.getByUsername(InitTestData.TEST_ADMIN_U
SERNAME), null, Lists.newArrayList(IdmAuthorityUtils.getAdminAuthority()),
"test");
    }

    @Before
    public void enableModule() {
        // enable example module
        moduleService.enable(ExampleModuleDescriptor.MODULE_ID);
    }

    @Test
    public void testPing() throws Exception {
        String message = "test";
        getMockMvc().perform(get(BaseController.BASE_PATH +
"/examples/ping")
                .with(authentication(getAuthentication()))
                .param("message", message)
                .contentType(InitTestData.HAL_CONTENT_TYPE))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.message", equalTo(message)));
    }
}
```