Process Automation & Security

Camunda Workshop Support

Camunda 8 presentation, Keycloak & Identity, Directory Server integration, Workers explained



Contents

[Revision History 3](#_Toc110406179)

[Introduction 4](#_Toc110406180)

[1) What is Camunda Platform 8.0? 4](#_Toc110406181)

[2) Business & IT collaboration across the lifecycle 5](#_Toc110406182)

[3) Camunda Cloud components 5](#_Toc110406183)

[4) Resilience and scalability 6](#_Toc110406184)

[5) Monitoring 6](#_Toc110406185)

[6) Reporting 7](#_Toc110406186)

[7) Reusability 8](#_Toc110406187)

[8) Camunda architecture 9](#_Toc110406188)

[First subject – Security 10](#_Toc110406189)

[1) Security Management within Camunda 8 10](#_Toc110406190)

[2) Local setup 10](#_Toc110406191)

[3) Configuring Keycloak 10](#_Toc110406192)

[4) Configuring Idenity 12](#_Toc110406193)

[5) Configuring the Spring Boot application 14](#_Toc110406194)

[a) Accessing TaskList and Operate APIs 14](#_Toc110406195)

[b) Securing the Spring Boot application through Keycloak 14](#_Toc110406196)

[c) Et voilà! 17](#_Toc110406197)

[2nd topic – TaskList & Operate API integration 18](#_Toc110406198)

[Explanations 18](#_Toc110406199)

[Some examples 18](#_Toc110406200)

[3rd topic - BPMN best practice 19](#_Toc110406201)

[4th topic – worker implementation 19](#_Toc110406202)

[1) Synchronous worker with variables as a POJO 20](#_Toc110406203)

[2) Synchronous worker with variables as a Map 20](#_Toc110406204)

[3) Synchronous worker with isolated variables 21](#_Toc110406205)

[4) Asynchronous worker with variables as a Map 21](#_Toc110406206)

# Revision History

#### 

|  |  |  |
| --- | --- | --- |
| **Date** | **Comment** | **Name** |
| August 01, 2022 | Document initialization | v1 |
| August 02, 2022 | Describing Camunda architecture and key features | v2 |
| August 02, 2022 | Explaining security | v3 |
| August 03, 2022 | Explaining APIs and workers | V4 |

# Introduction

## What is Camunda Platform 8.0?

Camunda Platform 8 is a cloud-native process automation tech stack designed for performance, throughput, and resiliency.

Camunda Platform 8 builds on the platform's open architecture and developer-friendly roots with a solution that delivers on the company’s vision of orchestrating anything and automating everything at scale; No matter if a managed cloud service is preferred (Camunda 8 SaaS) or the platform is going to be hosted in a private environment (Camunda 8 self-managed).

Camunda Platform 8’s new and improved workflow engine Zeebe is instrumental in Camunda’s achievement of universal process orchestration. Zeebe enables a SaaS offering such that teams can focus on process automation and not time consuming and expensive infrastructure setup, maintenance, and boilerplate code. Camunda Platform SaaS leverages Platform 8’s ability to scale with consistently low-latency and advanced fail-over capabilities across multiple regions including: US East, US West and Europe.

The new cloud native workflow engine in Camunda Platform 8 also unlocks native support for External Task Client Pattern which is the foundation for our all-new Connector ecosystem.

Camunda Platform 8’s Connector ecosystem vision is set to bring a new level of productivity to process automation by streamlining communication to:

* Cloud native applications such as Kafka, AWS Lambda functions, or Eventbridge.
* Technology enablers such as RPA, AI, or IoT
* A number of CRM, ERP and industry specific applications
* Messaging & productivity applications for quick notifications or to kick off a workflow
* Enterprise content & document management systems
* Data connectors for the pushing or pulling of data from BI systems, data lakes, or data warehouses

REST & SendGrid connectors are available, with an exciting roadmap to come.

Camunda Platform 8 is offered as SaaS or self-managed. Our SaaS offering is hosted on Google Cloud, while self-managed can be run on internal infrastructure or any cloud provider such as Google Cloud, AWS, or Microsoft Azure.

## Business & IT collaboration across the lifecycle

**Diagram

Description automatically generated**

Camunda enables your IT & Business teams to collaborate on creating process diagrams through an easy-to-use modeler. Once these diagrams have been created and approved, your developers can add technical functionality and bring them to life.

You can then deploy and run these processes from the Cloud Console or clients embedded into your custom code.

Other tools part of our software offering allow for conveniently viewing and managing process instances, as well as accessing detailed analytics to help your organization identify and eliminate operational bottlenecks. These analytics are designed to be manipulated by Business Analysts and Developer Citizens.

## Camunda Cloud components

Development in Camunda consists of the following phases and applications

* Phase 1, design and implement the processes

**Modeler** is used to model business process diagrams, decision tables, and user forms in collaboration between business and IT, adding automation properties and deploying the processes. It is available via the web and as a desktop application.

* Phase 2, execute the processes

**Zeebe** is the workflow engine. It is a newly designed cloud-native engine that is a result of 6 years of research and development. It uses a state-of-the-art event-based architecture that distributes and replicates the runtime state of your process instances robustly over several nodes. This architecture allows for high resilience and high scalability even in multi-region set-ups.

**Tasklist** is a web application that allows users to work on the BPMN user tasks assigned to them. You can use it as-is or build your own front-end on top of its comprehensive GraphQL API of tasklist

* Phase 3, Operate the system

**Operate** is a robust tool for monitoring and troubleshooting your live process instances.

* Phase 4, Analyze and improve your business

**Optimize** is our component for reporting and analytics. It is a self-service tool for business analysts who can create reports and dashboards on their processes

## Resilience and scalability

Zeebe is based on an event-based architecture that distributes and replicates the runtime state of your process instances robustly over several nodes.

Diagram

Description automatically generated

Your business logic lives in workers that interact with the Zeebe engine. The workers can be implemented in your favorite programming language since we have clients for all major languages. You can easily spin up and down workers to cover the current load.

The workers will send commands to Zeebe leveraging the fast and efficient gRPC protocol. The connection between the workers and the Gateway is a 2 ways persistent connection, optimal for high performance. Commands will then be handled as events on the Zeebe brokers’ partitions. This is where the BPMN logic is executed. Partitions are replicated and distributed over several cluster nodes.

Chart, line chart

Description automatically generatedIf a node is failing, partition leaders will be reelected to the remaining brokers to ensure service continuity. If your cluster is not able to handle the load, you can increase the number of partitions and brokers.

In contrast to classical approaches that often have performance bottlenecks in the persistence layer, Zeebe’s speed increases linearly with the size of the cluster. This means that you can build a cluster that handles your workload no matter how high it is

## Monitoring

Operate is the Camunda component built for monitoring and troubleshooting your live process instances.

Graphical user interface

Description automatically generated with medium confidence

It allows you to quickly see the instances with incidents, marked in red, and the exact process flow that each instance is on. From there you can bulk recover instances that have common errors (for example, after the connection to a third-party service has been reestablished) or recover them manually (and correct the process variables if required).

Operate comes with some pre-built filters to target more precisely instances that you’re looking at: Process, process version, execution dates, variables values, instance state (active, with incidents, completed, etc.).

## Reporting

Optimize is the Camunda component that brings you advanced analytics capabilities. You can create your own dashboards and reports or use some of the predefined ones at your convenience.

Here is an example of a simple dashboard:

Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generated

You can see the number of instances of a process, the average time, etc. The heatmaps are providing you very useful information: The first one (duration heatmap) gives you a direct vision on your process's bottleneck. The second one (frequency heatmap) gives you a direct vision on the more common process flow.

You can also add your own reports inside this dashboard, based on your process data. This could be pie charts with customer age distribution. This could be bar charts with monthly incomes ranges, etc.

Finally, it’s important to note that your dashboards can be filtered (on business data, for example per country) and that they can be exposed (prefiltered or not) to external users.

## Reusability

Developer friendliness is at the core of Camunda values. And as such, reusability is a must have. Reusability of:

* processes through call activities

You can develop processes that can be instantiated independently (from a website) or by other processes (through call activities). It could be for example a “document request” process, useful on its own, called during a customer registration, an incident resolution, etc.

* custom components as workers and connectors

Your processes will send notifications to your customers. This could be achieved by using the Sendgrid connector provided out of the box by Camunda. But you could also develop your own worker/connector and reuse it for any notifications sent from your processes.

* your current software factory: CI/CD, code quality, development languages

You can write your workers and UI in any language. If your preferred language is Java and that your tooling is around Jenkins and SonarQube, you can use them with Camunda. If your preferred language is Python and your CI/CD is Github actions, you can do the same with Camunda. You can reuse your existing IDE, software factory and best practices.

## Camunda architecture

Here is a picture showing a Zeebe Camunda Cloud Cluster in more detail.

Diagram

Description automatically generated

On the left, you see the different kinds of workflow client connections you can establish through a secure gRPC connection.

On the right, you’ll see a streaming exporter that hands data over real time to Elasticsearch, which acts as a data source for our other tools like Operate and Tasklist.

Everything runs on Kubernetes which is great for decoupling and high availability.

# First subject – Security

The 2 first topics discussed during our last call were “Authentication and Authorization using identity” and “LDAP integration”. We will handle both in that chapter, first discussing about LDAP integration and then about identity.

## Security Management within Camunda 8

Camunda 8 can be installed from docker-compose files or helm charts. These setups come with Identity and Keycloak integrated. We could decide to not use these components and to manage LDAP integration directly from the TaskList, Operate and Optimize applications (<https://docs.camunda.io/docs/1.3/self-managed/operate-deployment/authentication/>). But for this workshop, we decided to use this strategy because:

* It benefits from Keycloak features
* No modifications of the docker-compose or helm charts provided by Camunda
* Configurations are done once for all applications

## Local setup

* Apache Directory Studio running on localhost:10389 (no Active Directory available).
* Camunda 8.0.4 : <https://raw.githubusercontent.com/camunda/camunda-platform/d7848cc66f4dee79ee1ea73efa7eb9684c2dc748/docker-compose.yaml>
* Spring-Boot Project : <https://github.com/camunda-consulting/camunda-8-keycloak>

## Configuring Keycloak

Keycloak is an identity and access manager that can also be used to federate users from LDAP softwares as Active Directory. Locally, we have federated users from the Apache Directory but configurations steps are the same :

Under “User Federation”, it requires to add an LDAP provider

Graphical user interface, application

Description automatically generated

In the next screen, we will configure the federation. Some important configurations are :

* Username LDAP attribute: the LDAP attribute that will be mapped as Keycloak username. With Active directory, you would probably change the value to **'cn'.** Same for the RDN LDAP attribute.
* UUID LDAP attribute: Name of LDAP attribute used as unique object identifier in LDAP. For Active directory it should be **'objectGUID'**.
* User Object Classes: LDAP user records are found just if they contain all those object classes.
* Connection URL: your Active Directory url. As Camunda is running inside Docker containers and the Apache directory running on a local machine, the value is here host.docker.internal.
* Users DN: Full DN of LDAP tree where users are. It could be for example 'ou=users,dc=example,dc=com' assuming that your typical user will have DN like 'uid=john,ou=users,dc=example,dc=com'
* Bind DN: the system user DN used to connect to the LDAP provider
* Bind Credential: the the system user password

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

The LDAP provider can be tested (Test connection, Test authentication) and saved. It then requires to add the proper roles to the synchronized users. Depending on their groups or some attributes, you may want to give them access to Optimize, TaskList and/or Operate. To do so, we will use the Mappers. For this workshop, we have used hardcoded role mappers but you should use dynamic ones.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

One these configurations are done, you can access TaskList

## Configuring Idenity

In order to connect our Spring Boot application to TaskList APIs or Operate APIs, we will need to create an application in identity. To do so, we access Identity through the demo/demo user (we could use a synchronized user if he has the Identity Role) and we add an application:

Graphical user interface, text, application, email, website

Description automatically generated

Graphical user interface, text, application

Description automatically generated

And we then add permissions to access the APIs :

Graphical user interface

Description automatically generated

From the application details, we can the copy the client secret and use it in our code to access the APIs.

## Configuring the Spring Boot application

The Spring Boot application is an example of the custom code you would normally write around Camunda 8. It brings some Rest Services and a front-end. The Rest Services (backend) communicate with Zeebe through the Zeebe Client and with TaskList and Operate APIs.

### Accessing TaskList and Operate APIs

This application will interrogate the TaskList and Operate APIs. You can code everything from scratch or use existing community’s libraries:

<https://github.com/camunda-community-hub/camunda-tasklist-client-java>

<https://github.com/camunda-community-hub/camunda-operate-client-java>

In this project we have decided to use these libraries. We then must pass the ClientId and ClientSecret to the APIs client to start using them. In this project, the ClientId and ClientSecret are stored in the application.yaml file and passed to a Spring service that consumes the APIs via the clients. An example can be found here :

<https://github.com/camunda-consulting/camunda-8-complaint/blob/main/src/main/java/org/example/camunda/process/solution/service/TaskListService.java>

### Securing the Spring Boot application through Keycloak

For commodity reasons, we will use the same Realm that Camunda 8 is using, Camund-platform. We will create a new Client inside this Realm named SpringBootClient. It will be public and we will configure the valid redirect URLs as follow :

Graphical user interface, application

Description automatically generated

Configure the Valid redirect URLs

Graphical user interface, application

Description automatically generated

Now that that all configurations are done on the Keycloak side, we will configure the Spring Boot application. We will first add two dependencies. Indeed, the Spring Security is optional but brings some commodities to have fine grained authorization control in our Controllers (through annotations).

<!-- Keycloak -->

<dependency>

<groupId>org.keycloak</groupId>

<artifactId>keycloak-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

We will then configure our SpringBoot application to load the Keycloak properties from the application.yaml file rather than in a Keycloak.json :

@SpringBootApplication

**public** **class** ProcessApplication {

**[…]**

@Bean

**public** KeycloakSpringBootConfigResolver keycloakConfigResolver() {

**return** **new** KeycloakSpringBootConfigResolver();

}

}

We can then define our Keycloak properties in our **application.yaml** file. The resource is the client name we defined in Keycloak. The principal attribute will be the Keycloak attribute that we will use as username:

keycloak.auth-server-url: http://localhost:18080/auth

keycloak.realm: camunda-platform

keycloak.resource: SpringBootClient

keycloak.public-client: true

keycloak.principal-attribute: preferred\_username

We will finally create a new SecurityConfiguration that will :

* allow us to annotate rest endpoints with required roles (attributes of the EnableGLobalMethodSecurity)
* define KeycloakAuthenticationProvider as the authentication provider (method configureGlobal)
* is orientied towards human users : RegisterSessionAuthenticationStrategy
* require to be authenticated to access any resource (method configure)

@KeycloakConfiguration

@EnableWebSecurity

@Configuration

@EnableGlobalMethodSecurity(prePostEnabled = **true**,

securedEnabled = **true**,

jsr250Enabled = **true**)

@ComponentScan(basePackageClasses = KeycloakSecurityComponents.**class**)

**public** **class** WebSecurityConfig **extends** KeycloakWebSecurityConfigurerAdapter {

@Autowired

**public** **void** configureGlobal(AuthenticationManagerBuilder auth) {

SimpleAuthorityMapper grantedAuthorityMapper = **new** SimpleAuthorityMapper();

KeycloakAuthenticationProvider keycloakAuthenticationProvider = keycloakAuthenticationProvider();

keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(grantedAuthorityMapper);

auth.authenticationProvider(keycloakAuthenticationProvider);

}

@Bean

@Override

**protected** SessionAuthenticationStrategy sessionAuthenticationStrategy() {

**return** **new** RegisterSessionAuthenticationStrategy(**new** SessionRegistryImpl());

}

@Bean

@Override

@ConditionalOnMissingBean(HttpSessionManager.**class**)

**protected** HttpSessionManager httpSessionManager() {

**return** **new** HttpSessionManager();

}

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

**super**.configure(http);

http.authorizeRequests()

.antMatchers("/admin/\*").hasRole("Admin")

.anyRequest().authenticated()

.and().csrf().disable();

}

}

### Et voilà!

We now have a Spring Boot application that is fully secured through Keycloak within the same Realm used by Camunda. We can now add some role based security layer to the endpoints. This can be done in the configure method of the previous class, or it can be done through annotations:

@PreAuthorize("hasRole('Tasklist')")

@GetMapping("/myArchivedTasks")

**public** List<Task> getCompletedTasks(@PathVariable String userId) **throws** TaskListException {

String username = getAuthenticatedUser();

**return** taskListService.getAssigneeTasks(username, TaskState.***COMPLETED***, **null**);

}

We could even define custom annotations to bring some more conciseness:

@Target(ElementType.***METHOD***)

@Retention(RetentionPolicy.***RUNTIME***)

@PreAuthorize("hasRole(Admin)")

**public** **@interface** IsAdmin {

}

Also, in the controllers, we will require to identify the connected user. This could be done simply that way :

@Autowired

**private** HttpServletRequest request;

**protected** String getAuthenticatedUser() {

**return** getKeycloakSecurityContext().getIdToken().getGivenName();

}

**private** KeycloakSecurityContext getKeycloakSecurityContext() {

**return** (KeycloakSecurityContext) request.getAttribute(KeycloakSecurityContext.**class**.getName());

}

2nd topic – TaskList & Operate API integration

## Explanations

Operate API is based on Rest and fully documented here :

<https://docs.camunda.io/docs/apis-clients/operate-api/>

TaskList API is based on GraphQL and fully documented here :

<https://docs.camunda.io/docs/apis-clients/tasklist-api/tasklist-api-overview/>

These APIs require an authentication to be accessed. This mechanism for the self managed C8 has been described in the previous chapter about security. To gain some time, if developing in Java, I would recommend using community clients that handle the different authentication mechanisms. As these projects are community based, any contribution is welcome.

## Some examples

To list all tasks assigned to Arminius, you could write something like :

LocalIdentityAuthentication sa = **new** LocalIdentityAuthentication().clientId("XXX").clientSecret("XXX");

CamundaTaskListClient client = **new** CamundaTaskListClient.Builder().shouldReturnVariables().taskListUrl("http://localhost:8082/").authentication(sa).build();

List<Task> tasks = client.getAssigneeTasks("Arminius", TaskState.***CREATED***, **null**);

To list all process definitions of the process “Hello World”, you could write something like :

LocalIdentityAuthentication sa = **new** LocalIdentityAuthentication().clientId("XXX").clientSecret("XXX");

CamundaOperateClient client = **new** CamundaOperateClient.Builder()

.operateUrl("http://localhost:8081/").authentication(sa)

.build();

ProcessDefinitionFilter processDefinitionFilter = **new** ProcessDefinitionFilter.Builder().name("Hello World").build();

SearchQuery procDefQuery = **new** SearchQuery.Builder().withFilter(processDefinitionFilter).withSize(20)

.withSort(**new** Sort("version", SortOrder.***ASC***)).build();

List<ProcessDefinition> processDefinitions = client.searchProcessDefinitions(procDefQuery);

3rd topic - BPMN best practice

Camunda Academy is offering free access to BPMN courses : <https://academy.camunda.com/camunda-bpmn>

Graphical user interface, website

Description automatically generated

4th topic – worker implementation

Workers are very flexible. You can write synchronous workers, asynchronous workers, work with process variables as Maps or as POJO. Let see different worker implementations and what they are best suited for :

## Synchronous worker with variables as a POJO

@ZeebeWorker(type = "my-service", autoComplete = **true**)

**public** ProcessVariables invokeMyService(@ZeebeVariablesAsType ProcessVariables variables) {

***LOG***.info("Invoking myService with variables: " + variables);

**Long** result = myService.myOperation(variables.getTexte());

**return** **new** ProcessVariables().setOperationResult(result);

}

This worker will call spring service operation and send the operation result as a new process variable named “OperationResult”. The service is synchronous because of the autoComplete attribute of the ZeebeWorker annotation. It’s well suited if myService.myOperation is not taking too much time.

Working with process variables inside a POJO enforce consistency. If for any reason, someone considers changing process variables, compilation will ensure that the worker is updated accordingly.

## Synchronous worker with variables as a Map

@ZeebeWorker(type = "generatePdf", autoComplete = **true**)

**public** Map<String, Object> archiveAuditTrail(**final** ActivatedJob job) **throws** IOException, XDocReportException, InterruptedException {

Map<String, Object> variables = job.getVariablesAsMap();

String targetFileName = (String) variables.get("fileName");

DocxPdfUtils.*generatePdf*((String) variables.get("templateName"), targetFileName, variables);

Drive drive = DriveUtils.*drive*();

// upload to Google Drive

File fileMetadata = **new** File();

fileMetadata.setName(targetFileName);

fileMetadata.setMimeType("application/pdf");

InputStreamContent mediaContent = **new** InputStreamContent("application/pdf", **new** FileInputStream(target));

File file = drive.files()

.create(fileMetadata, mediaContent)

.setFields("id")

.execute();

**return** Map.*of*("driveId", file.getDriveId());

}

This worker will generate a Pdf document from a Docx template, using the variables to resolve the velocity placeholders. It will then store the file in Google Drive and store the driveId in a process variable named driveId.

This worker can be used in any process and therefor, we don’t know what is the structure we will receive. That’s why working with variables as a Map is the best option. Generating a PDF and storing it into Drive is not too long and we can keep this worker synchronous (autocomplete = true)

## Synchronous worker with isolated variables

@ZeebeWorker(type = "email", autoComplete = **true**, forceFetchAllVariables = **true**)

**public** ProcessVariables sendEmail(

ActivatedJob job,

@ZeebeVariable String to,

@ZeebeVariable String bcc,

@ZeebeVariable String subject,

@ZeebeVariable String template) **throws** MessagingException, IOException {

***LOG***.info("Sending email to " + to + " and bcc " + bcc + " using template " + template + " and subject: " + subject);

Map<String, Object> variables = job.getVariablesAsMap();

User consultant = userService.getUserByUsername((String) variables.get("consultant"));

variables.put("consultant", consultant);

mailService.sendMail(to, **null**, bcc, subject, template, variables);

**return** **new** ProcessVariables(); // new object to avoid sending unchanged variables

}

This worker is a generic worker that can be used within multiple processes. But it’s slightly different than the previous one because we always expect to have variables to, bcc, subject and template available (in the previous one, we had no expectations). That’s why we have these @ZeebeVariable String to as parameters.

## Asynchronous worker with variables as a Map

**private** **void** sendMail(MimeMessagePreparator messagePreparator, JobClient client, ActivatedJob job) {

mailSender.send(messagePreparator);

client.newCompleteCommand(job.getKey()).send().exceptionally((throwable -> {

**throw** **new** RuntimeException("Could not complete job", throwable);

}));

}

@ZeebeWorker(type = "sendMeetingReport")

**public** **void** sendMeetingReport(@ZeebeVariablesAsType Meeting meeting, JobClient client, ActivatedJob job) **throws** Exception {

MimeMessagePreparator messagePreparator = mimeMessage -> {

MimeMessageHelper messageHelper = **new** MimeMessageHelper(mimeMessage, **true**);

messageHelper.setFrom(from);

};

**new** Thread(() -> sendMail(messagePreparator, client, job)).start();

}

This worker is asynchronous. It was considered that the SMTP connection could take too much time and the job completion is executed once the mail is sent. A better example could be to handle some asynchronous service calls with response sent over a MOM queue.