
Flowable
making process personal

Flowable Trainings

Developing for Flowable Platform

Introduction

Your new best friend

Your Trainer:

Roger Villars
Solution Architect

Contact us:
training@flowable.com
https://forum.flowable.com/

Other resources:
https://docs.flowable.io

Organization

First things first!

Social

- First name basis

Breaks

- One break per workshop, but tell us if you need more

Questions

- **Don’t mind to interrupt for questions – this course is meant to be interactive!**
- Longer / specific discussions during the break or in workshop

Objectives

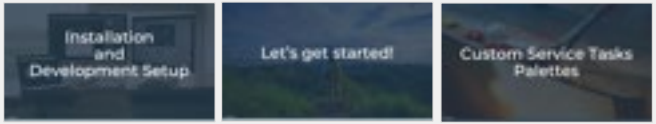
What you will know…

Remark: This training covers version 3.11.0

- **Understand the concepts**
  - Learn the basics of Flowable development
- **Learn how to set up and configure Flowable**
  - Set up your first Flowable Work application locally
  - Create a simple workflow with a service task
  - Make the service task customizable
- **Learn how to work with the Flowable APIs**
  - Start, query and inspect your Engines using Runtime, Repository, History and Management Services
  - Learn how to extend and integrate your Flowable Applications

Agenda

- First Day



- Second Day

- Third Day

Chosen Topics on request.


Installation
and
Development Setup

Prerequisites

What you need to bring!

- Proposed Flowable Work Project, on version 3.11:
  - Java Development Kit (JDK) 8 or **AdoptOpenJDK 11** (Hotspot)
  - Maven repository with **Flowable artifacts** configured in Maven's settings.xml
  - A Java IDE (we recommend **IntelliJ IDEA**)

- **Licenses** for all Flowable products

Initializr Project

Your template code is waiting for you!

- We prepared a basic custom project for you!
- Demo setup created with the Flowable Initializr.
- A step-by-step guide to integrate similar repos can be found here:

  https://documentation.flowable.com/latest/develop/be/java-extensions/#create-a-custom-project
- http://initializr.flowable.io/
- https://github.com/flowable/flowable-training-getting-started

People / Matthias Stöckli / Dashboard / Matthias Stöckli's Home

Flowable Training

Repository can be found here:

https://github.com/flowable/flowable-training-getting-started

Project:

Unzip this project anywhere on your dev machine.

License:

Copy this file to C:\Users\%userprofile%\.flowable (folder .flowable may not exist yet)

Settings XML:

Copy this file to C:\Users\%userprofile%\.m2 (folder .m2 may not exist yet)

Project Structure

- Multi-module Maven project with parent pom.xml
- Maven submodules for
  - Flowable Work
  - Flowable Design
  - Flowable Control
- Local settings.xml needed to access the Flowable Maven Repository (`~/.m2/settings.xml`)

Spring Boot

- Flowable is based on Spring Boot
- Start web applications as a simple Java Process (no fat EE application servers).
- Improved maven dependency management with starter dependencies.
- Autoconfiguration based on dependencies.
- Dependency Injection for Service classes
- Defined configuration management with @Configuration classes and application.properties/yml files.

Spring Boot

Almost no code !?

- Main entry point for any Spring Boot application is the Application class.
- Usually annotated with `@SpringBootApplication`
- Starts up the full Flowable application provided by dependencies.
- Check the training project:
  `com.training.handson.design.TrainingHandsonDesignApplication`
  `com.training.handson.work.TrainingHandsonWorkApplication`
  `com.training.handson.control.TrainingHandsonControlApplication`
- Additional classes to define project specific configuration (e.g. Security) annotated with `@Configuration`

Configuration Properties

Configure Flowable the “Spring Boot Way”

- You can configure most things in an `application.properties/yaml` file.
- Located in `src/main/resources`
- Special property files can be activated via Spring Boot profiles (`spring.profiles.active=<profile>`)
- All **Flowable properties** start with ”`flowable.`”
- You can find a reference for all properties in the **online documentation**.

Infrastructure

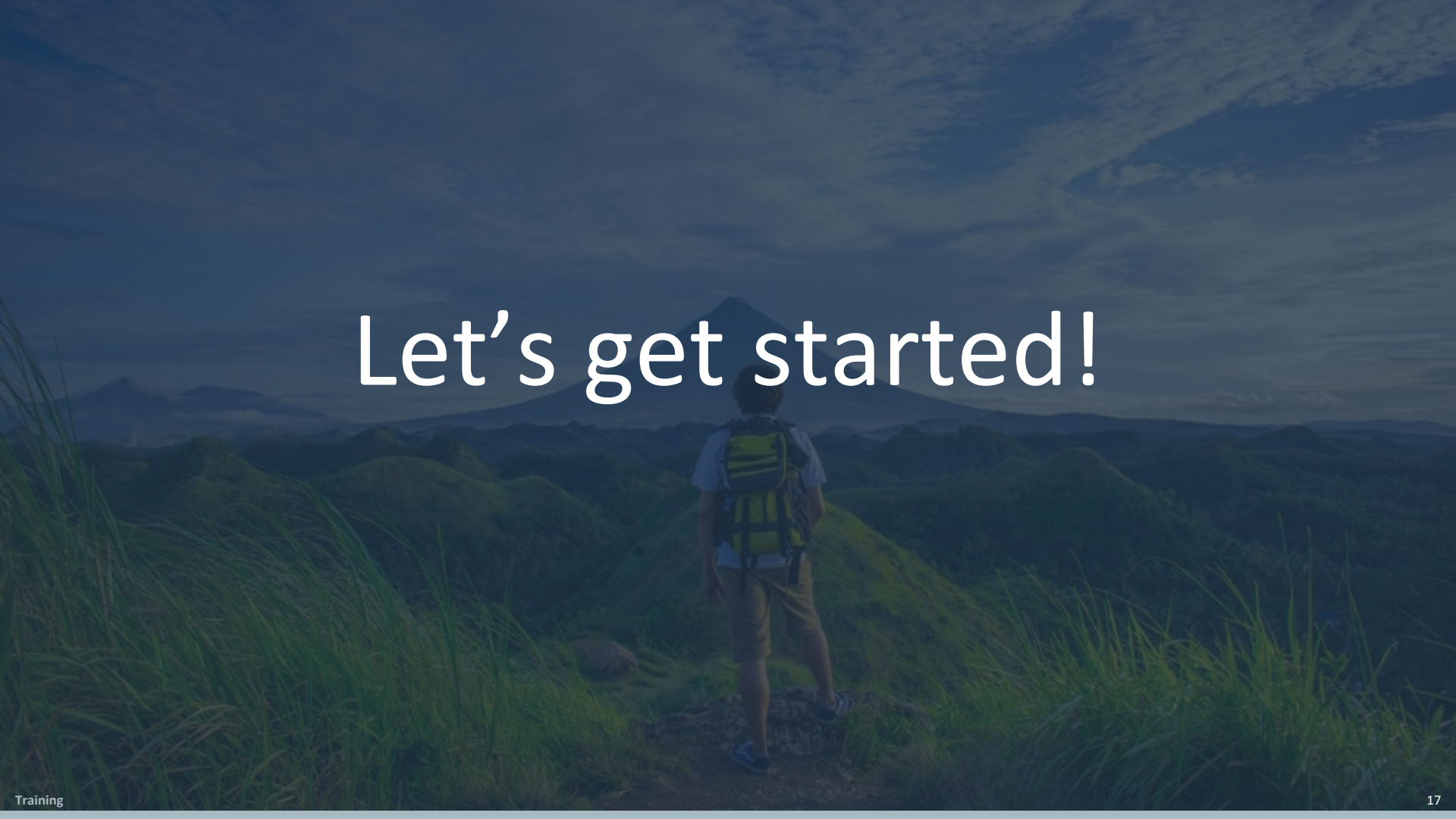
- Move your license file to `~/.flowable/flowable.license`

Without Infrastructure:

- No additional Infrastructure needed
- Enable the `infraless` Spring profile for all modules.
- Embedded H2 database, location: `~/flowable-data/training-handson`
- No Elasticsearch needed, direct DB access for queries
- **This is only for demo and training purposes, do not use this in production!**

With Infrastructure:

- Download and Install Docker **OR**
- Download and Install Postgres and Elasticsearch 7.12.x


Let’s get started!

Create a Service Task

Why a Service Task?

- Flowable Work offers rich **out-of-the-box feature set**

- Not everything can be done “no-code”
  - Sometimes **domain specific logic** is required
  - No API available (e.g., REST, Event, …)
  - Access to Flowable API
  - Execute code snippets

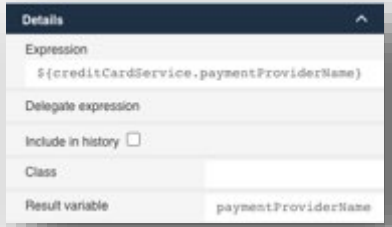
Service Tasks

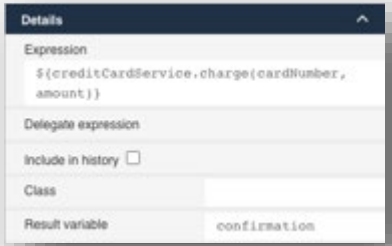
- Service tasks represent **work done by a computer.**
- Usually, it calls **Java code** and can involve external systems.
- There are four ways of declaring the Java logic to invoke:
  - Evaluating a **value expression**
  - Invoking a **method expression**
  - Class extending an **AbstractPlatformTask**
  - Evaluating an **expression** that resolves to a delegation object

Service Task JUEL Expressions

- Evaluating a value expression

```
<serviceTask id="serviceTask1" name="Get Payment Provider"
flowable:expression="${creditCardService.paymentProviderName}"
flowable:resultVariable="paymentProviderName" />
```





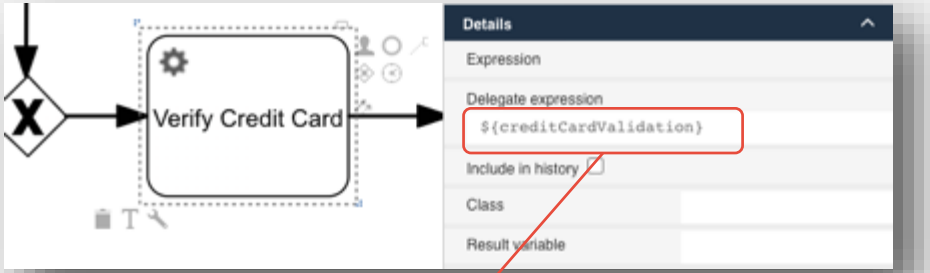
- Invoking a method expression

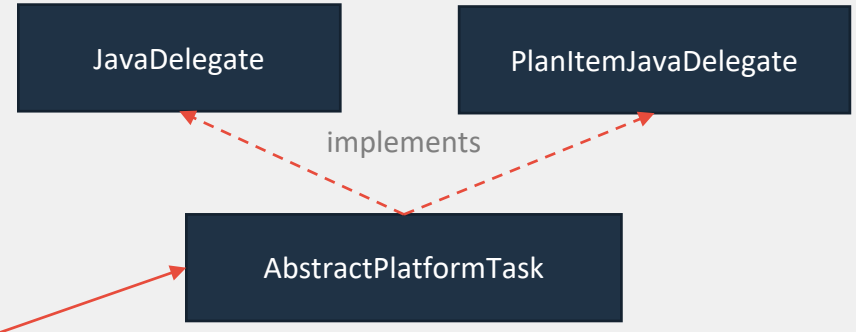
```
<serviceTask id="serviceTask2" name="Charge Credit Card"
flowable:expression="${creditCardService.charge(cardNumber, amount)}"
flowable:resultVariable="confirmation" />
```

- Evaluating an expression that resolves to a delegation object

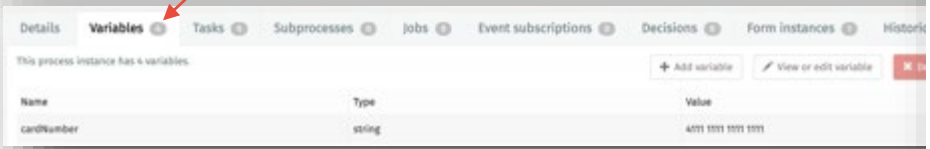
```
<serviceTask id="serviceTask3"
flowable:delegateExpression="${creditCardValidation}" />
```

Using a Delegate Expression



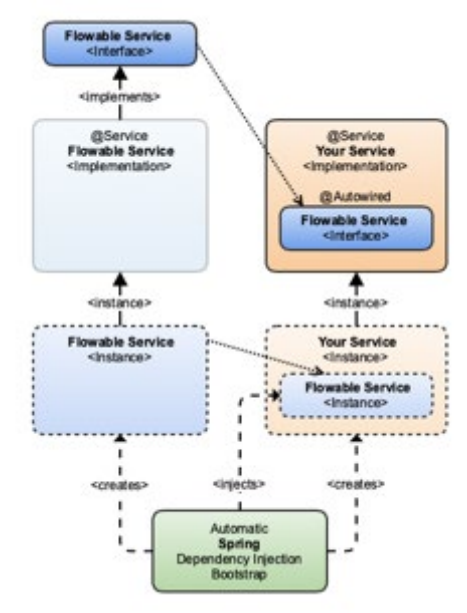



```
@Component("creditCardValidation")
public class CreditCardValidationTask extends AbstractPlatformTask {

    @Override
    public void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer) {

    }

}
```




```
public interface VariableContainer {

    boolean hasVariable(String variableName);

    Object getVariable(String variableName);

    void setVariable(String variableName, Object variableValue);

    void setTransientVariable(String variableName, Object variableValue);

    String getTenantId();

}
```

Dependency Injection



- Flowable uses the Spring Dependency Injection mechanism.
- You do not need to instantiate Flowable services by your own if you want to use them in your service (annotated with **@Service/@Component**).
- Just add the interface of the Flowable service and annotate it with **@Autowired**.
- Spring will then inject the Flowable service instance at runtime automatically.

Variable Container

- Used to **manage** Variables in the Current Context

- A variable has a **name,** a **value**, a **scope** and a **type**:
  Name: `"cardNumber"` Value: `"4111 1111 …"`
  Scope: `"PRC-1234-2452-1234"` Type: `"string"`

- There are **different** variable containers:
  - Process Instances
  - Case Instances
  - Tasks
  - …

```
public interface VariableContainer {

    boolean hasVariable(String variableName);

    Object getVariable(String variableName);

    void setVariable(String variableName, Object variableValue);

    void setTransientVariable(String variableName, Object variableValue);

    String getTenantId();

}
```

Variable Types

Variable Types define what your variables behave like.

Each variable has a **type** in Flowable, for instance:

- **String** Text values
- **Boolean** Boolean values
- **Integer** Numbers without fractions
- **Double** Numbers with fractions
- **Date, *Instant*** Dates including time (UTC)
- **LocalDate** Dates without time
- **JSON** Complex object stored as JSON

- **ContentItemVariable** Content Items
- **DataObjectVariable** Data Object references
- **Serializable (😈)** Binary content (use JSON instead)

It is possible to define your own types.

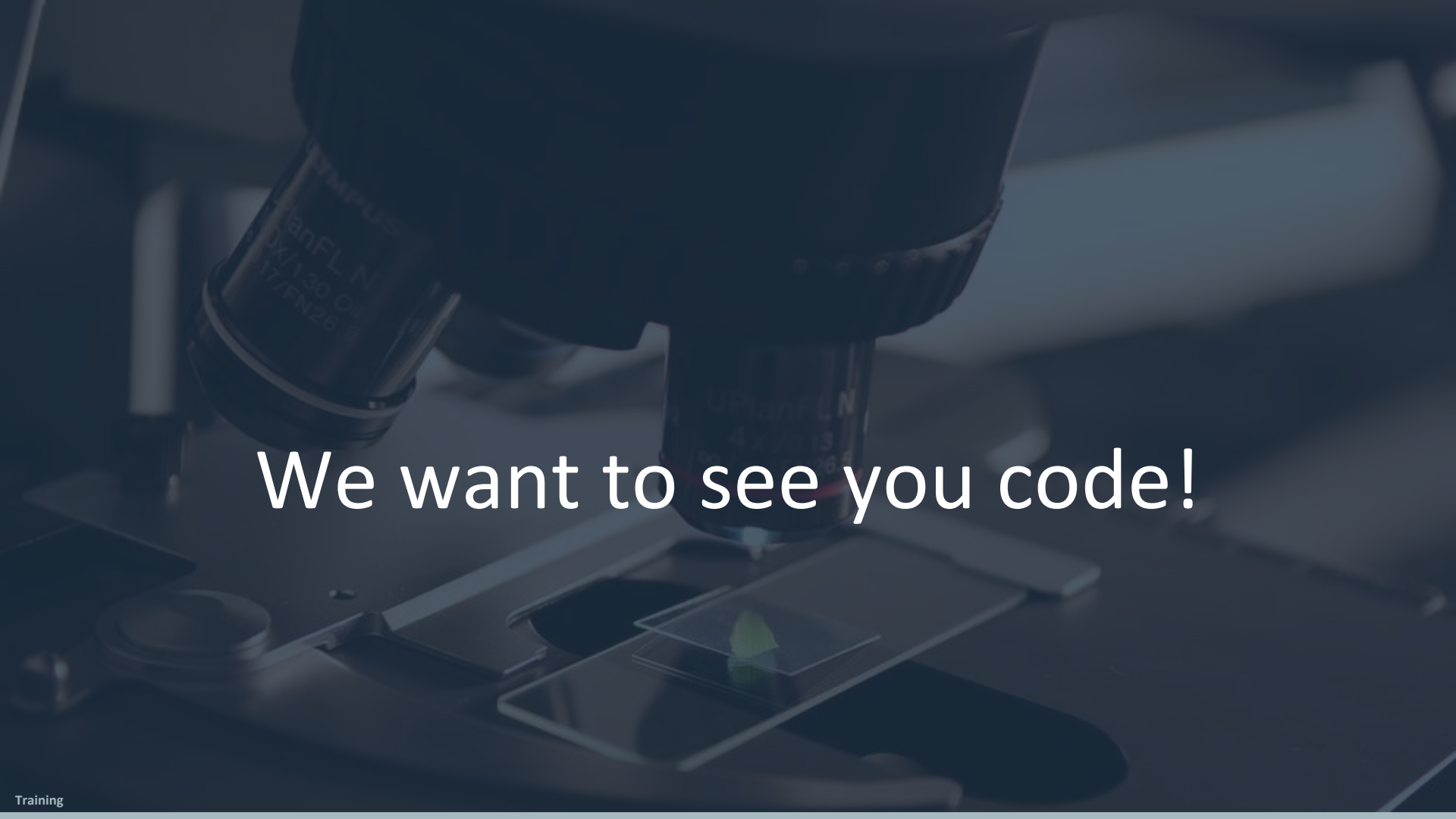
Hello World

Let’s do it the Flowable way.

Goal:

- Create a new process where a user can enter and store a **greeter** and a **person to be greeted**.
- greeter and personToBeGreeted are **Flowable users** (use the “Person” select form component).
- Use these variables in in a **Spring Service** to log a simple a text in the following form:
  <<greeter>> says: Hello <<personToBeGreeted>>
- **Advanced:** Maybe we can improve the service to log the “Display Name” of the users instead of their IDs.

  Hint: Inject the `PlatformIdentityService`!


We want to see you code!


Custom Service Tasks
Palettes

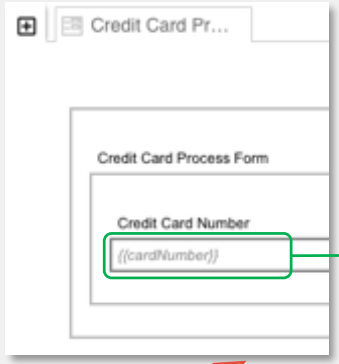
Recap and Motivation

- Service Tasks are there to execute **Java Code** in BPMN and CMMN

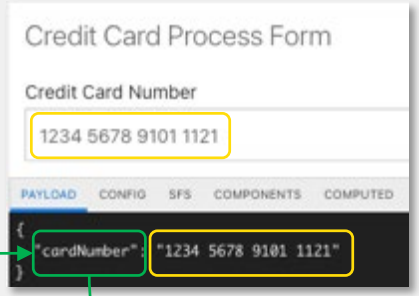
- Challenges
  - How to make them usable for **Citizen Developers**?
  - How to keep the **Process Knowledge outside** of the code?
  - How to make them reusable **across** different models?

Tightly coupled Model and Code

Model



Instance



Service Task

Tight coupling !!

Code

```
@Override
public void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer) {
    String cardNumberValue = (String)variableContainer.getVariable( "cardNumber");
```

Custom Service Tasks

- Custom Service Tasks are achieved by extending the Flowable Design **process and case palettes**.
- Palettes define **which elements** can be added to a BPMN process or a CMMN case.
- Add tasks **extending Service Tasks** with **delegateExpression** and **extensionElements** as arguments

Steps to add your own Service Task

Palette

1. Create a **new palette** if not already done.
2. Add a new component extending **ServiceTask**.
3. Fill the **delegateexpression** attribute with a reference to the bean implementing your Task behavior.
4. Hide and **add** attributes as needed.
5. Copy your palette into the **libs of your designer**.

Java Logic

6. Create a bean extending **AbstractPlatformTask.**
7. Read from **extensionAttributes**, **variableContainers** etc. and implement your custom business logic in **executeTask()**.

Abstract Platform Task

```
public abstract void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer);
```

Implements *JavaDelegate* (BPMN) and *PlanItemJavaDelegate* (CMMN) – can be used in both

- *getExtensionElementValue:* resolves a palette value
- *getStringExtensionElementValue:* same, but returns *String*
- *getExtensionElements:* complex items with multiple elements
- *resolveValue:* resolves a value based on provided vars

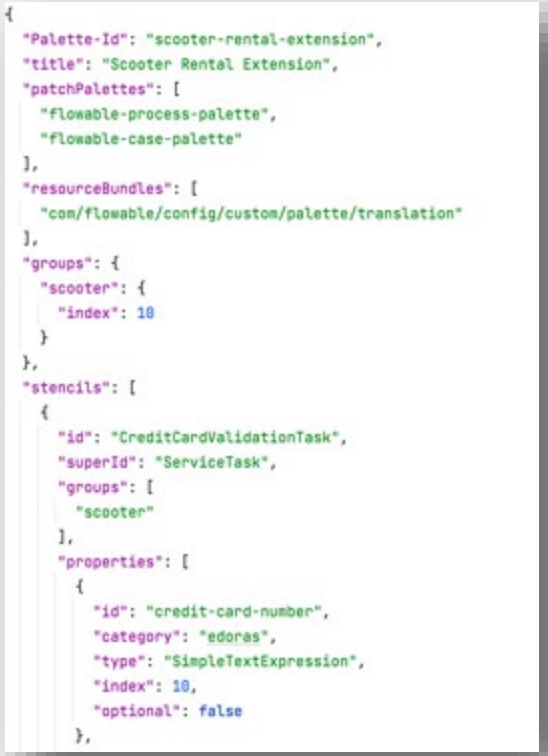
Customize the Palette

- Create a Custom Flowable Design project

```
<dependency>
    <groupId>com.flowable.design</groupId>
    <artifactId>flowable-spring-boot-starter-design</artifactId>
    <version>${flowable-design.version}</version>
</dependency>
```

Palette File

- Format XML or **JSON**
- Location: **com/flowable/config/custom/palette**
- Filename: ***.palette**
- Pitfalls
  - JSON file endings are not supported
  - Resource Folders are displayed with dot instead of slash in IntelliJ

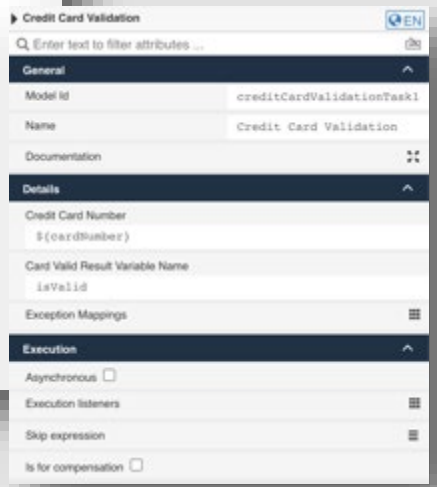


```
{
  "Palette-Id": "scooter-rental-extension",
  "title": "Scooter Rental Extension",
  "patchPalettes": [
    "flowable-process-palette",
    "flowable-case-palette"
  ],
  "resourceBundles": [
    "com/flowable/config/custom/palette/translation"
  ],
  "groups": {
    "scooter": {
      "index": 10
    }
  },
  "stencils": [
    {
      "id": "CreditCardValidationTask",
      "superId": "ServiceTask",
      "groups": [
        "scooter"
      ],
      "properties": [
        {
          "id": "credit-card-number",
          "category": "edoras",
          "type": "SimpleTextExpression",
          "index": 10,
          "optional": false
        },
```

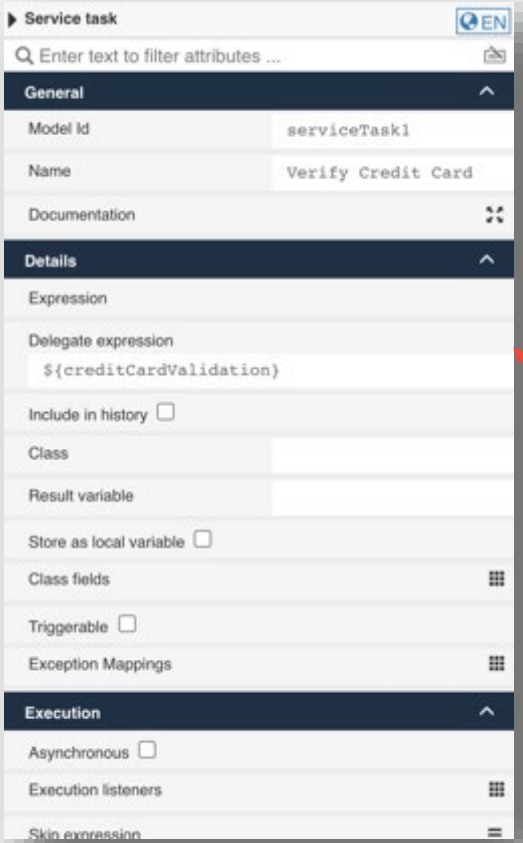
A Custom Palette Item

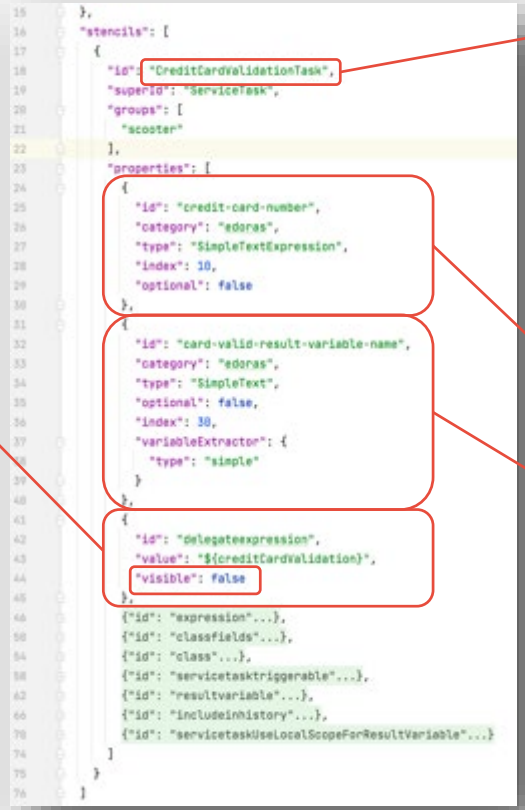
- **Metadata**:
  - ID of the Palette
  - Palettes to be patched
  - Resource bundles
- **Stencils**
  - Attributes
  - Your logic resides in the properties: “*expression*” or “*delegateexpression*”
- **JSON Schema**
  - https://documentation.flowable.com/latest/develop/json-schemas/

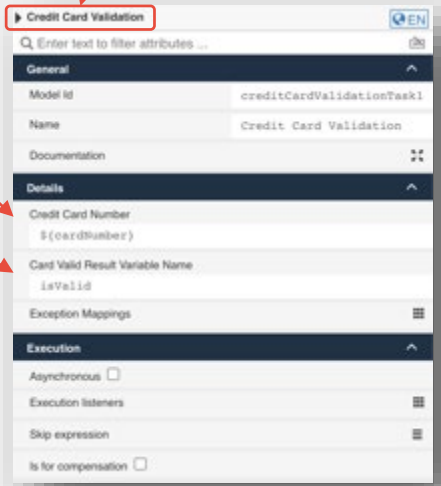
```
{
  "Palette-Id": "scooter-rental-extension",
  "title": "Scooter Rental Extension",
  "patchPalettes": [
    "flowable-process-palette",
    "flowable-case-palette"
  ],
  "resourceBundles": [
    "com/flowable/config/custom/palette/translation"
  ],
  "groups": {
    "scooter": {
      "index": 10
    }
  },
  "stencils": [
    {
      "id": "CreditCardValidationTask",
      "superId": "ServiceTask",
      "groups": [
        "scooter"
      ],
      "properties": [
        {
          "id": "credit-card-number",
          "category": "edoras",
          "type": "SimpleTextExpression",
          "index": 10,
          "optional": false
        },
```



Creating a Palette

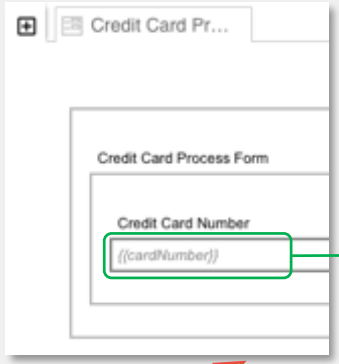




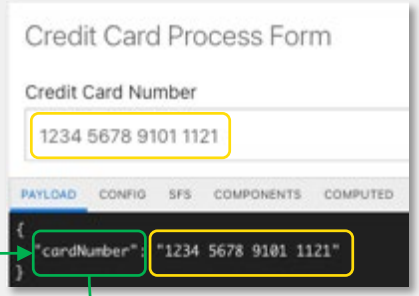


Tightly coupled Model and Code

Model



Instance



Service Task

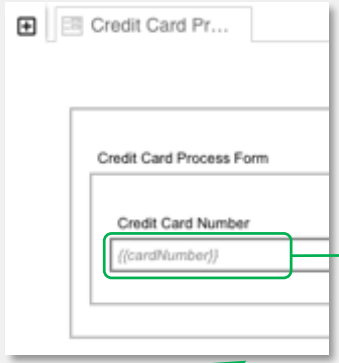
Tight coupling !!

Code

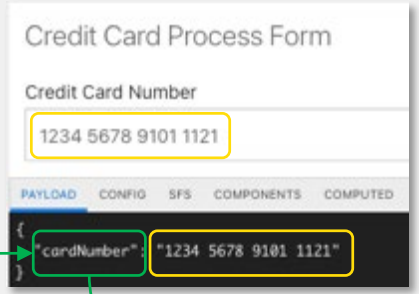
```
@Override
public void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer) {
    String cardNumberValue = (String)variableContainer.getVariable( "cardNumber");
```

Loosely coupled Model and Code

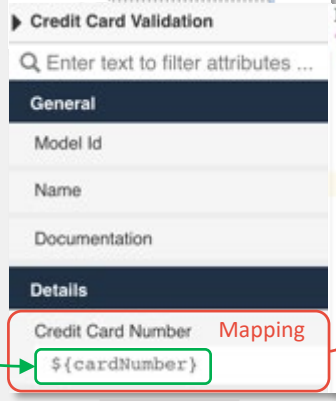
Model



Instance



Service Task **with ExtensionElement**



Loose coupling !!

Code

```
@Override
public void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer) {
    String cardNumberExpression = extensionElementsContainer.getExtensionElement( elementName: "credit-card-number").get().getText();
    String cardNumberValue = this.resolveValue(variableContainer, cardNumberExpression);
```

Flowable Out-of-the-box Palettes

- flowable-vis-palettes
  - com/flowable/vis/palette/core.process.palette `flowable-process-palette`
  - com/flowable/vis/palette/core.case.palette `flowable-case-palette`
  - com/flowable/vis/palette/form.template.palette
    - com/flowable/vis/palette/core.form.palette `flowable-form-palette`
    - com/flowable/vis/palette/core.page.palette `flowable-page-palette`
  - com/flowable/vis/palette/core.drd.palette `flowable-drd-palette`


Exercise and Demo!

Hello World

Let’s do it the Flowable way.

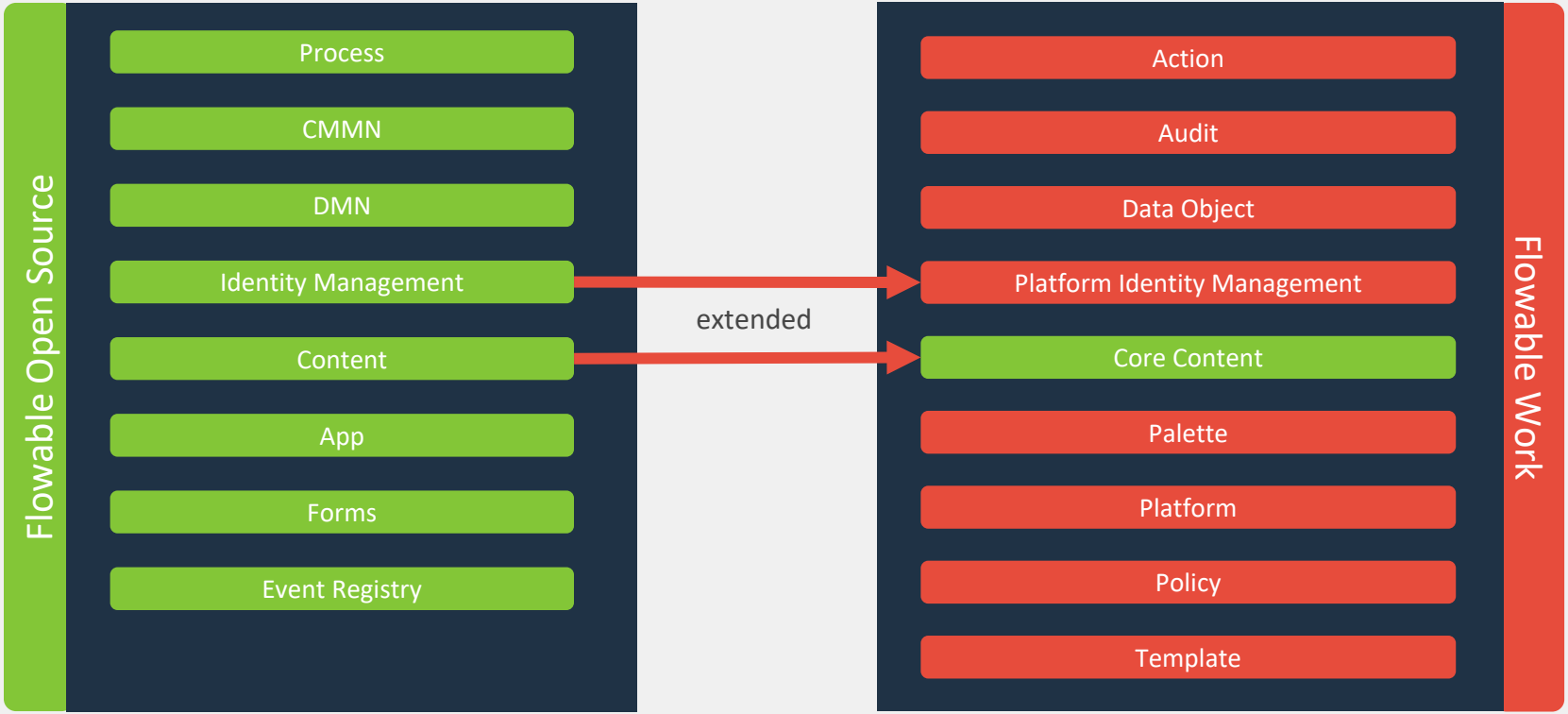
Goal:

- A stub for creating a Flowable Design palette extension has been provided to you. Add it to the Flowable Design code in **com/flowable/config/custom/palette.**
- Give the **delegateExpression** Attribute a fixed value, so it always call **${greeterService}.** Make the attribute read only.
- Add 2 additional **SimpleTextExpression** attributes to the new palette entry:
  - greeterVariable
  - toBeGreetedVariable
- Adapt the GreeterService code so, that the variables containing greeter and personToBeGreeted are now defined by Flowable Design.

Hint: Use this.getStringExtensionElementValue()

Flowable API

List of Engines



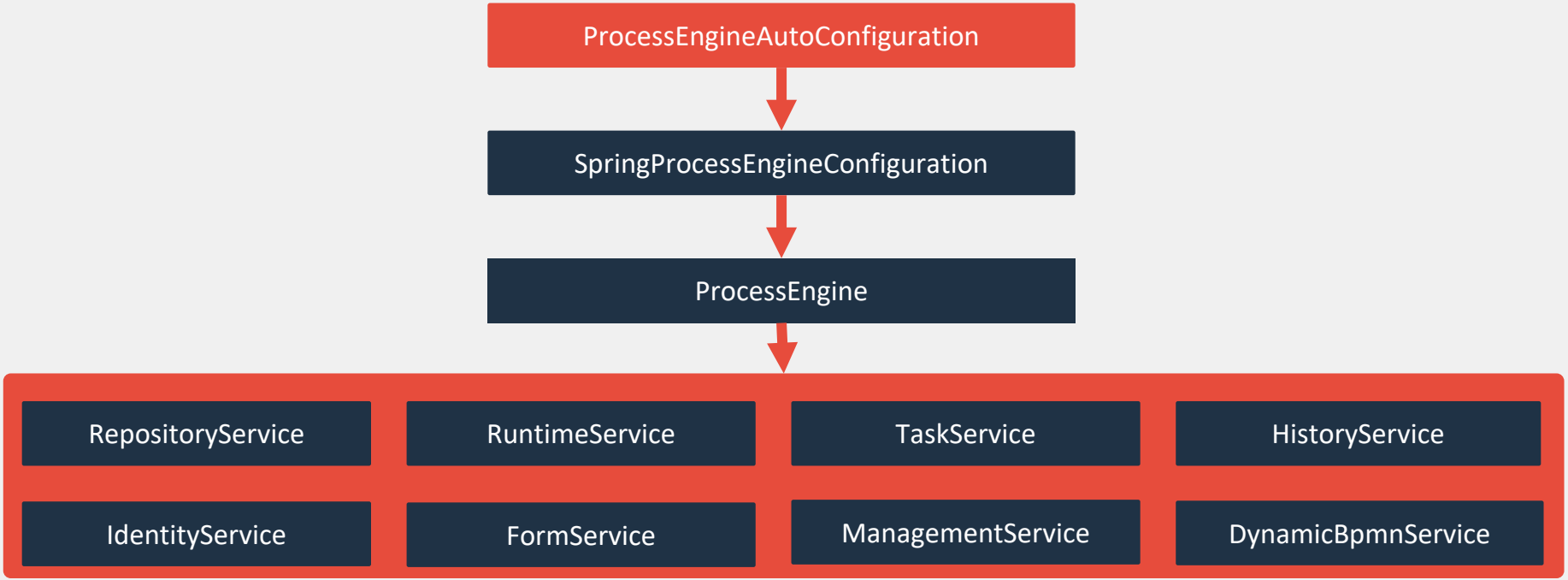
Engine structure

- Engines are created from an **Engine Configuration**

- Each engine provide different services. Examples:
  - **Repository**: manages models, deployments and definitions
  - **Runtime**: starts and controls instances
  - **History**: exposes all data gathered by the engine

- Services offer operations and queries
  - `runtimeService.startProcessInstanceByKey("holidayRequest", variables);`
  - `taskService.createTaskQuery().taskCandidateGroup("managers").list();`

Example: Platform Process Engine



BPMN Java API

Your entry to the BPMN engines.

- **RuntimeService:**
  - Start, complete and query **Process Instances**
  - Start, complete and query **Executions**
  - Set and delete **variables**
  - Add and delete **identity links**
  - Get, add and delete **entity links**
- **RepositoryService:**
  - Create, search and delete **deployments and definitions**
- **HistoryService:**
  - Create and query **historic entities**
- **ManagementService:**
  - Move, delete and query for **jobs**
  - Gain **low-level insights** on table structure etc.

BPMN REST API

/process-api/{service}/{operation}/{id?}

Example: GET Endpoints

All Instances:	/process-api/runtime/process-instances
Single Instance:	/process-api/runtime/process-instances/{PROCESS_ID}
Definitions:	/process-api/repository/process-definitions
Single Definition:	/process-api/repository/process-definitions/{DEFINITION_ID}
Deployments:	/process-api/repository/deployments
Single Deployment:	/process-api/repository/deployments/{DEPLOYMENT_ID}

Important BPMN Tables / Naming

- ACT_RE_*:
  Repository. Static information such as process definitions and resources.

- ACT_RU_*:
  Runtime. Data of living instances, tasks, jobs…

- ACT_HI_*:
  History. Historic data of past processes.

- ACT_GE_*:
  General Data, other tables such as the “ByteArray” table

CMMN Java API

There are a few essential Java Services.

- **CmmnRuntimeService:**
  - Start, complete and query **Case Instances**
  - Start, complete and query **PlanItems**
  - Set and delete **variables**
  - Add and delete **identity links**
  - Get, add and delete **entity links**
- **CmmnRepositoryService:**
  - Create, search and delete **deployments and definitions**
- **CmmnHistoryService:**
  - Create and query **historic entities**
- **CmmnManagementService:**
  - Move, delete and query for **jobs**
  - Gain **low-level insights** on table structure etc.

CMMN REST API

/cmmn-api/{service}/{operation}/{id?}

Example: GET Endpoints

All Instances:	/cmmn-api/cmmn-runtime/case-instances
Single Instance:	/cmmn-api/cmmn-runtime/case-instances/{CASE_ID}
Definitions:	/cmmn-api/cmmn-repository/case-definitions
Single Definition:	/cmmn-api/cmmn-repository/case-definitions/{DEFINITION_ID}
Deployments:	/cmmn-api/cmmn-repository/deployments
Single Deployment:	/cmmn-api/cmmn-repository/deployments/{DEPLOYMENT_ID}

Important CMMN Tables

- `ACT_CMMN_RU_*:`
  Runtime. Data of living case instances, plan items and sentries.

- `ACT_CMMN_HI_*:`
  History. Historic data of case instances and plan items.

- `ACT_CMMN_CASEDEF:`
  Case definitions

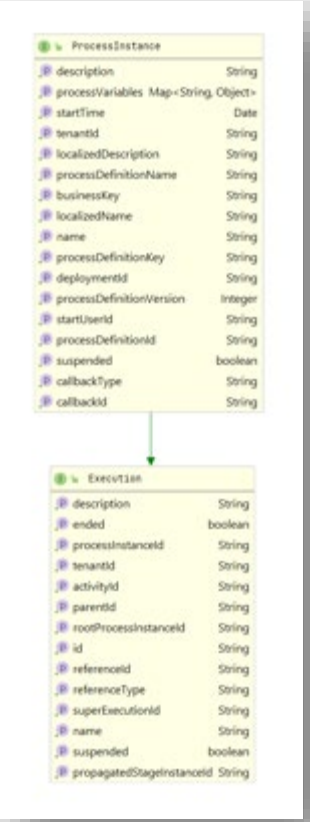
- `ACT_CMMN_DEPLOYMENT:`
  CMMN Deployments and resources

Most important terms

- **Process Model:** BPMN definition of a business process.
- **Process Instance**: Representation of a concrete process. It contains variables for storing the information.
- **Activity**: Each of the steps of a process model representing ‘work’ to be done: User Task, Service Task, Script Task.
- **Execution**: Represents a path of execution of a process instance. A process instance can have several executions.
- **Wait State**: Process state where a process is waiting for an event to occur.
- **Variable:** Information unit stored in a process instance.

Process Instances

- In Flowable, you will usually interact with **process instances**.
- They hold together all executions and act as «**root execution**» of a process themselves. More on that later.
- Each process instance has:
  - A start and end date
  - A unique **processInstanceId** (starting with PRC-)
  - A **process definition** key and ID
  - A **name**, **description** and **business key**



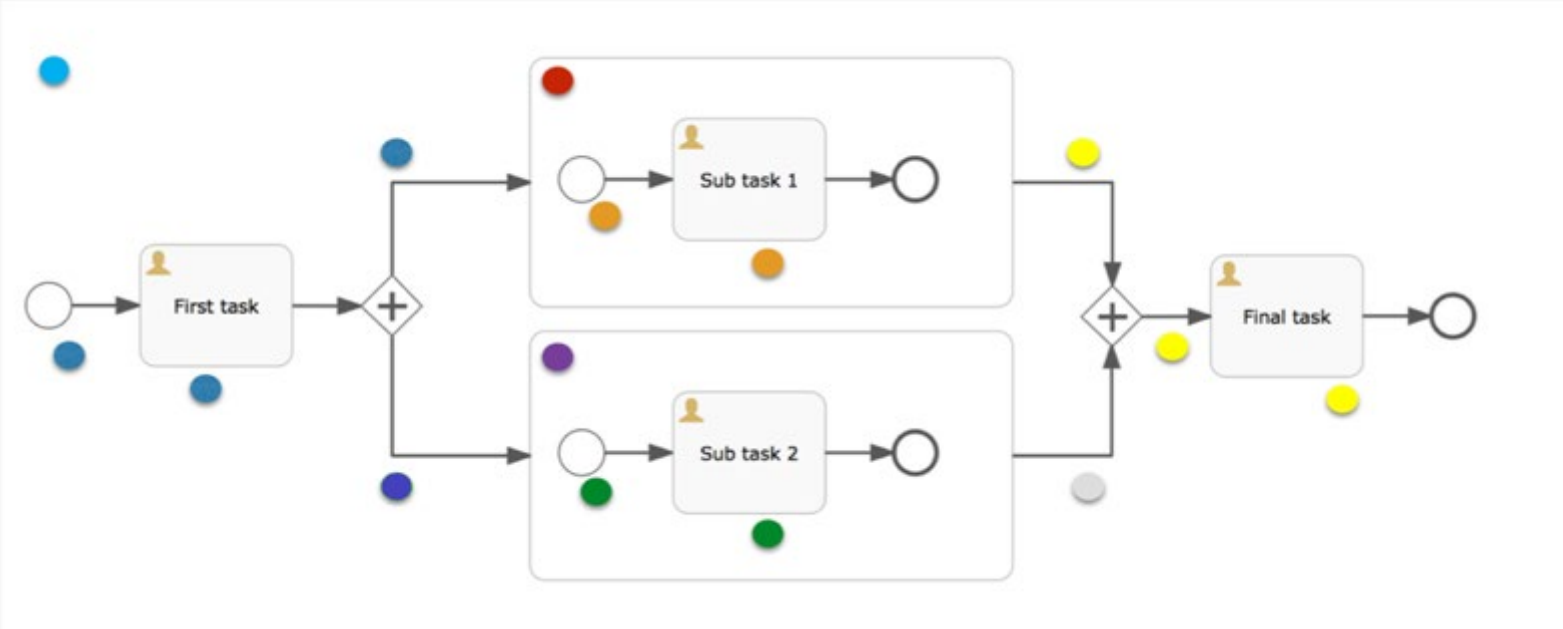
Executions

Find your path with executions!

- A process instance consist of a tree of **executions**. The root process is an execution itself.
- The process Workflow is performed in a new execution.
- For alternative paths and **subProcesses** new executions are created.
- A sub process has a **scoped execution**, which means that they each have their own variables.
- The current execution is always available in expressions: **${execution.id}**

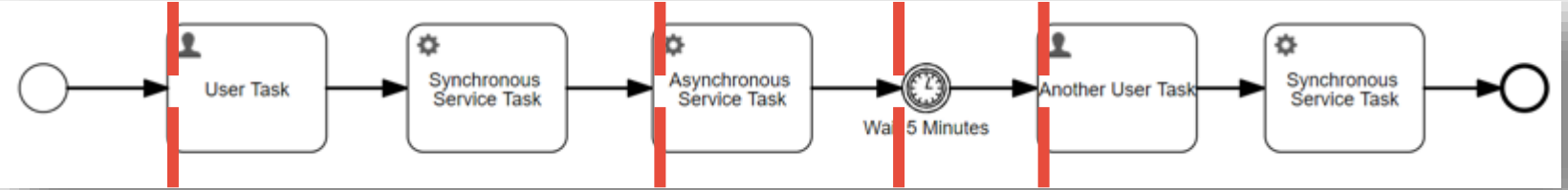
Execution tree

Look at the code in t60-bpmn-technical-foundations to see the tree in action



Wait state

- Most activities are executed **synchronously**, **in sequence** in a **unique transaction**.
- Process steps are executed until a **wait state** is reached or the process is completed.
- Wait states are most often introduced by **User Tasks**, **Timer Events** and **asynchronous tasks**.
- If an error arises during the execution, **the whole transaction is rolled back**, including all the steps since the previous wait state



Synchronous vs Asynchronous

***Store order* Synchronous**

- If *Store Order* fails, *Enter Order is also rolled back* and the order is lost.

***Store order* Asynchronous**

- *Enter Order* is committed before processing *Store Order.*

Most important terms

- **Case Instance**: Representation of a concrete case. Contains variables and has a state.
- **Plan Item**: Tasks, Event Listeners, Stages and Milestones
- **Plan Item Instance**: A concrete instance of a plan item, e.g. a task. If the ”Repetition” rule is active, a plan item can have multiple instances.
- **State**: The state of Case Instances and Plan Items are predefined by their lifecycle defined in the CMMN specification.
- **Transition**: When a plan item or case instance changes from one state to another, this is called a “Transition”.

CMMN in a Nutshell

Modeling Cases

- CMMN is used to model **cases**
- There are no **Start** and **End Events**, instead, the state of the whole model is taken into account
- **Plan Items** define your case:
  - **Tasks** define what is happening in a case
  - **Event Listeners** react to an external event
  - **Milestones** define that a certain state was achieved
  - **Stages** are groups of above elements
- Each **Plan Item** has a **state** and can **transition**
- **Sentries** define criteria to enable (enter) or terminate (exit)
- **Connectors** are used to connect PlanItems.

Variables in CMMN

- Variables are stored on a **Case Instance**.
- Since CMMN models are constantly re-evaluated, you must be mindful about **variable initialization**.
- Every condition of an entry or exit sentry will be **re-evaluated on each cycle** – use the null-safe variable function delegates if you want to check for variable values:
  `${var:equals("yourVariable", "someValue")}`


Variables and Expressions

Flowable Data

Store all your data!

- In Flowable, when dealing with cases or processes, we store and access information in **variables**.
  You do not directly access the Flowable DB.

- A variable has a **name,** a **value**, a **scope** and a **type**:
  Name: `"city"` Value: `"Valencia"`
  Scope: `"PRC-1234-2452-1234"` Type: `"string"`

- There are a number of **variable containers**:
  Process instances, case instances, tasks etc.

- **Not everything is a variable though!**
  Some fields such as IDs, names, business keys etc. are “**native fields**”.

Example: Data in Flowable

Case: Car Rental Case

`"id": "CAS-4fd3a362-84bb-412a-9738"`	Native Field
`"name": "Annie Austin's Card Rental Contract"`	Native Field
`"businessKey": "REN-2009-5231"`	Native Field
`"carType": "Audi A8"`	String Variable
`"startDate": "2019-02-14"`	Date Variable
`"startMileage": 9001`	Number Variable

Process: Damage Report Process

`"id":" "PRC-e5834f60-3ff6-4bc1-8235"`	Native Field
`"damagedParts": ["Driver Seat", "Window"]`	List Variable

```
"analysis": {
    "text": "Clearly something is broken."
    "estimatedCost" : 1200
}
```

JSON Variable

```
"someBinaryVariable":
"NUL ý NAK WÁ F3 NUL H ENQ EM <* NUL Hƒã(ÃÌÌÌÌHƒì(° DC3 NUL NUL NUL H
à7* NUL A* BS NUL BS NUL E3ÀèÂs DC3 NUL °# NUL NUL NUL H
õ7* NUL E3ÉE3Àè«s DC3 NUL ° DC3 NUL NUL NUL NUL H
SI8* NUL E3ÉE3Àè"s DC3 NUL ° ETX NUL NUL NUL NUL H
(8* NUL E3ÉÀ ¸Y ESC NUL NUL éz» DC3 NUL ° SOH NUL NUL NUL H
>8* NUL AƒÈyDJ DC3 èas DC3 NUL ° õ NUL NUL NUL NUL H
U8* NUL E3ÈAƒÈyèIs DC3 NUL ° SOH NUL NUL NUL NUL H
```

Binary Variable

Variable Types

Variable Types define what your variables behave like.

Each variable has a **type** in Flowable, for instance:

- **String** Text values
- **Boolean** Boolean values (true or false)
- **Integer, *Short, Long*** Numbers without fractions
- **Double** Numbers with fractions
- **Date, *Instant*** Dates including timezone.
- **LocalDate, *LocalDateTime*** Dates w/o timezones (e.g. birthdays)
- **JSON** Complex object stored as JSON

- **ContentItemVariable** Content Items (e.g. attachments)
- **DataObjectVariable** Data Object references
- **JPAEntityVariable** Reference to a JPA Entity
- **Serializable (😈)** Binary content (use JSON instead!)

It is possible to define your own types.

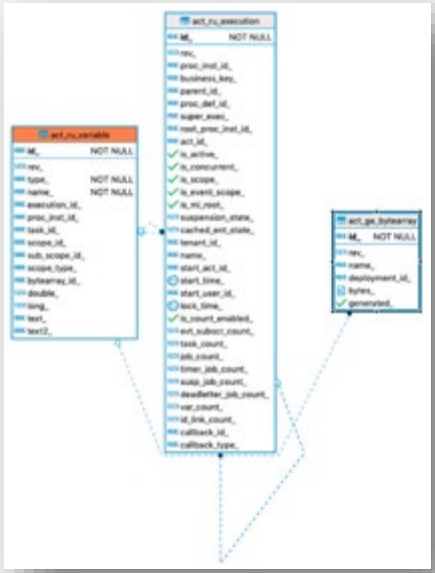
How to store variables

Forms and friends

- There are many ways to store variables, e.g.:
  - Through **Form Bindings**
  - As the result of **Service Tasks**
  - Through **Initialize Variables Tasks**
  - As part of some **backend logic**
  - As output of **DMN Decision Tables**
  - Through the **REST API**
- While possible, **avoid changing** the **type of variable** after you defined it!
- Variables can be **deleted** or set to «**null**»

Variable Storage (DB)

- Each variable is backed by a **variable instance** which is stored in **ACT_RU_VARIABLE** table.
- Depending on the type, the actual content is either stored in the **DOUBLE_**, **LONG_**, **TEXT_** column.
- More complex objects are stored in the table **ACT_RU_BYTEARRAY** table and referenced through the **BYTE_ARRAY_ID_**
- **You will usually not interact with data in the database directly.**



Setting Variables in code

- A process instance can have **variables** that are stored in the DB table ACT_RU_VARIABLE

- Add variables at **process creation** and **during process execution:**
  - `execution.setVariable("varName", "varValue");`
  - `variableContainer.setVariable("varName", "varValue");`

Variable scope

Store your variables locally.

- By default, variables are put on the **highest parent** when set:
  The process instance execution.

- Variables can be stored on the current **local** scope(execution) by calling the xxxLocal method.
  - `execution.setVariableLocal("myVar", "value");`

- Local variables are **only visible** in the current execution.

Transient Variables

- Transient variables behave like regular variables, but are not persisted.
  - `execution.setTransientVariable("varName", "value");`
- Only available until the **next wait state**.
- A transient variable shadows a persistent variable with same name:
  - `execution.getVariable("varName")`
- This will return the transient variable (if existing) or the regular variable.
- Supported by some tasks, e.g. the **HTTP Task or Service Task**.

Expressions (1/3)

- Flowable accepts **JUEL** expressions for getting variables values and calling methods:
  - `${myVar}`
  - `${myBean.doSomething(myVar, execution)}`

- Can be used in **ServiceTasks**, ExecutionListeners, conditions etc…
- Types:
  - Value expressions
  - Method expression
  - Expression functions

Expressions (2/3)

- **Value expressions** resolve to a value. All process variables and Spring beans are available:
  - `${myVar}`
  - `${myBean.myProperty}`
- **Method expressions** invoke a method. Parameters can be literals, expressions or special objects(execution,task, userId)
  - ${myBean.doSomething(myVar, execution)}

Expressions (3/3)

- **Expression functions** provide shortcuts to certain functionality, e.g. variables.
  - Retrieves a value in a null-safe way:
    `${variables:get("myVar")}`
  - Null-safe check for equality of two values:
    `${variables:equals("myVar" , value)}:`

    Many more…

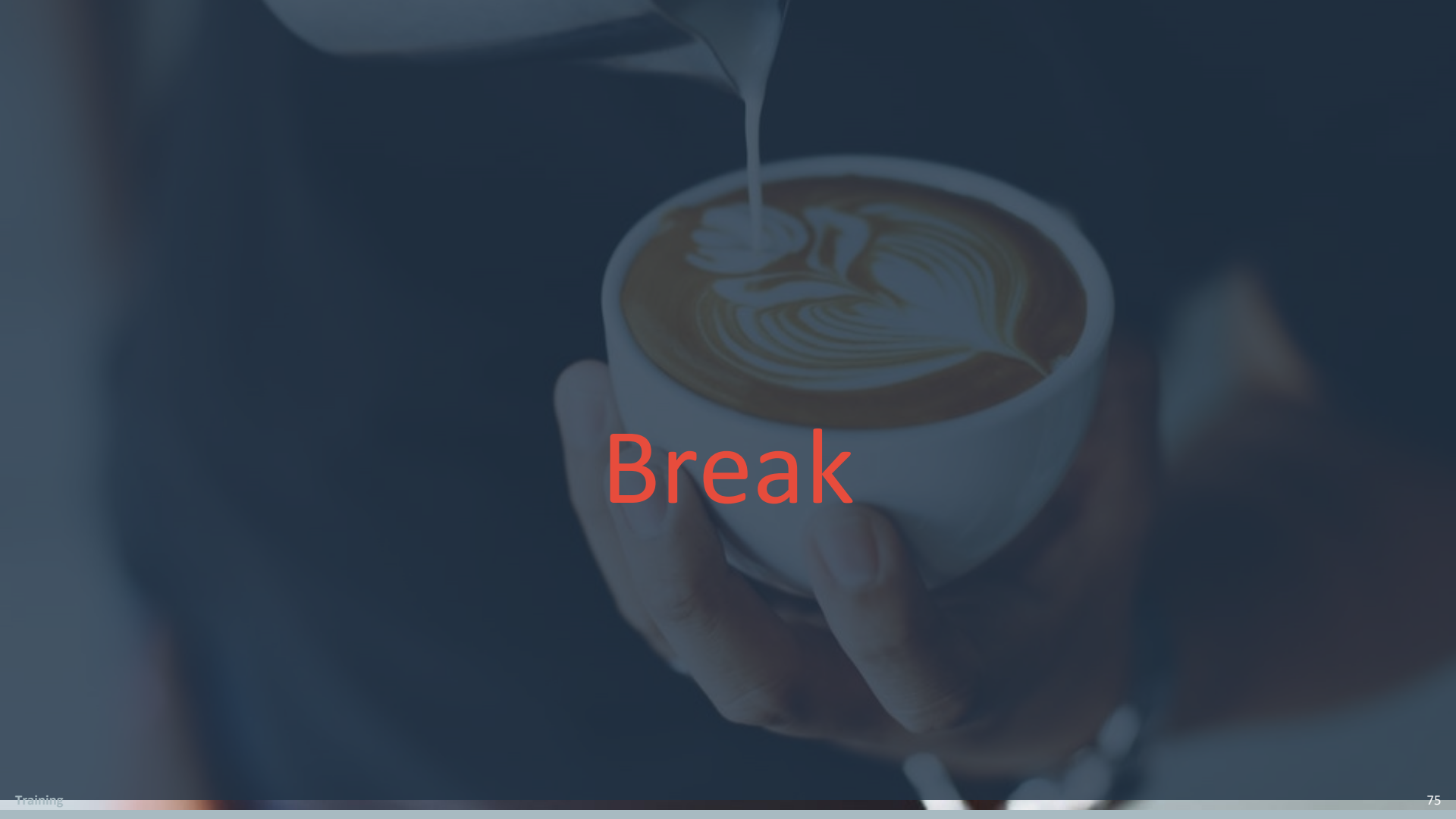
- **Aliases: `vars`** or **`var`** can be used instead of **`variables`**

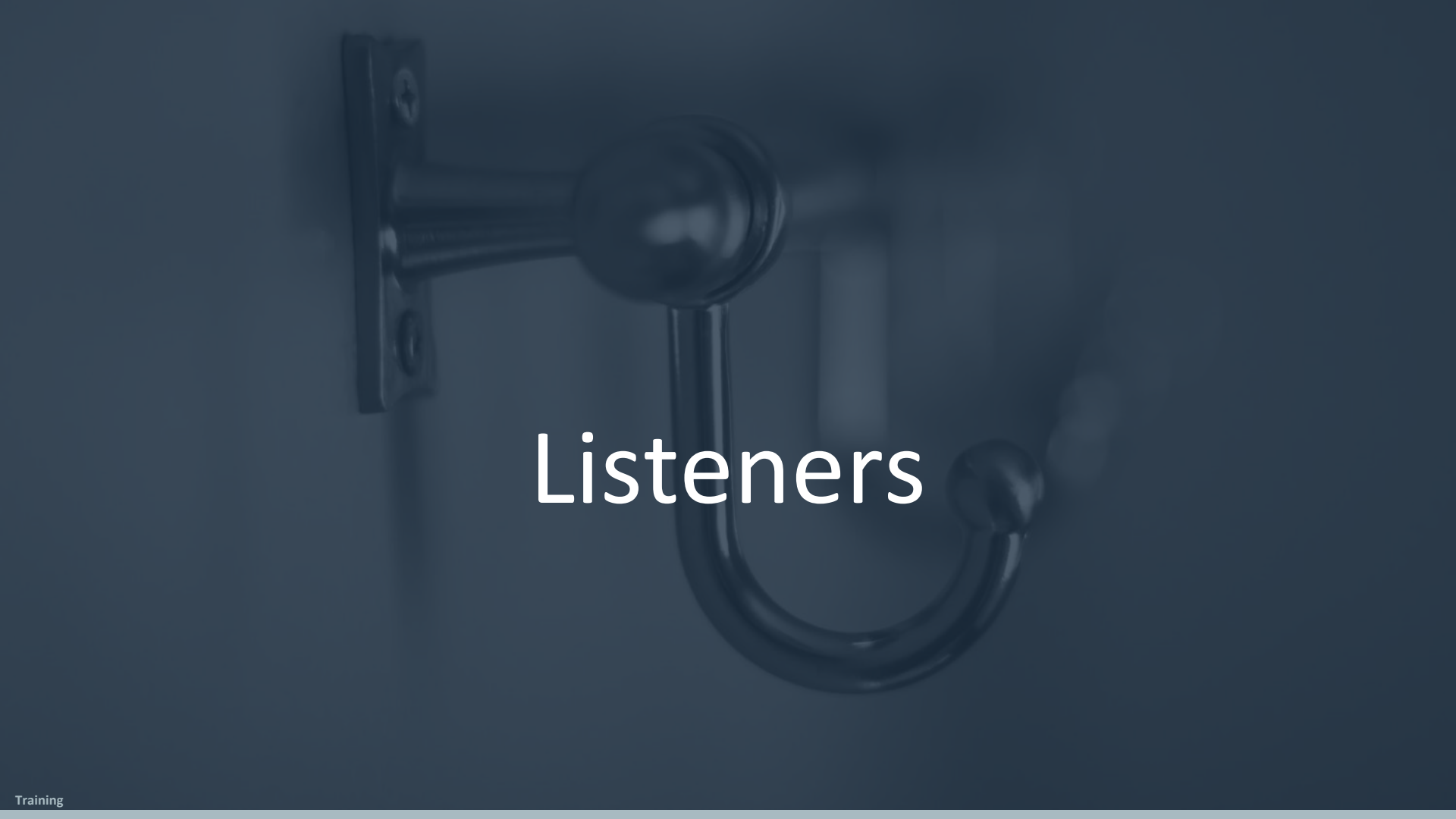
- The execution/case instance is automatically injected.

Flw Backend Expressions

Following backend Services examples are generic and already implemented in flw:

Topic	Service Expression	Description
timeUtils	${flwTimeUtils.now()}	Returns the current date and time in UTC.
CollectionUtils	${flwCollectionUtils.listWithElements()}	Return A new list with elements
MathUtils	${flwMathUtils.sum()}	Return the sum
StringUtils	${flwStringUtils.upperCase()}	Return String, converted to uppercase


Break


Listeners

Listeners

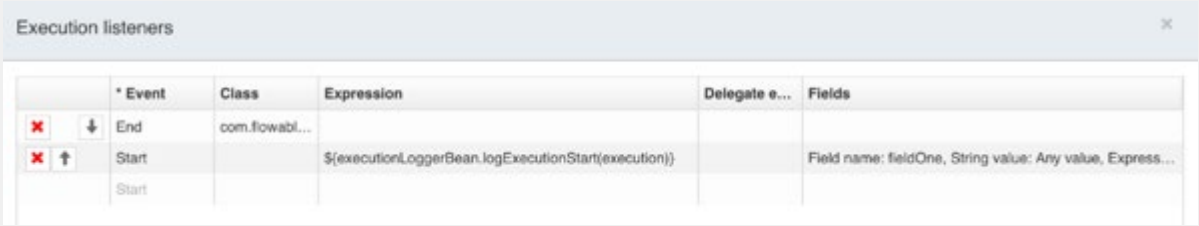
- Listen to **events** in processes or cases **declaratively**
- Motivation: Implement technical details without bloating the model with script or service tasks (depends on modelling granularity)
- **Execution Listeners (BPMN)**
  - Can be added to any element in the model
  - Events: Start, Transition, End
- **Task Listeners (BPMN and CMMN)**
  - Can be added to user tasks
  - Events: Create, Assignment, Complete, Delete
- **Lifecycle Listeners (CMMN)**
  - Can be added to any CMMN element, listens to state transitions

Listeners Implementation

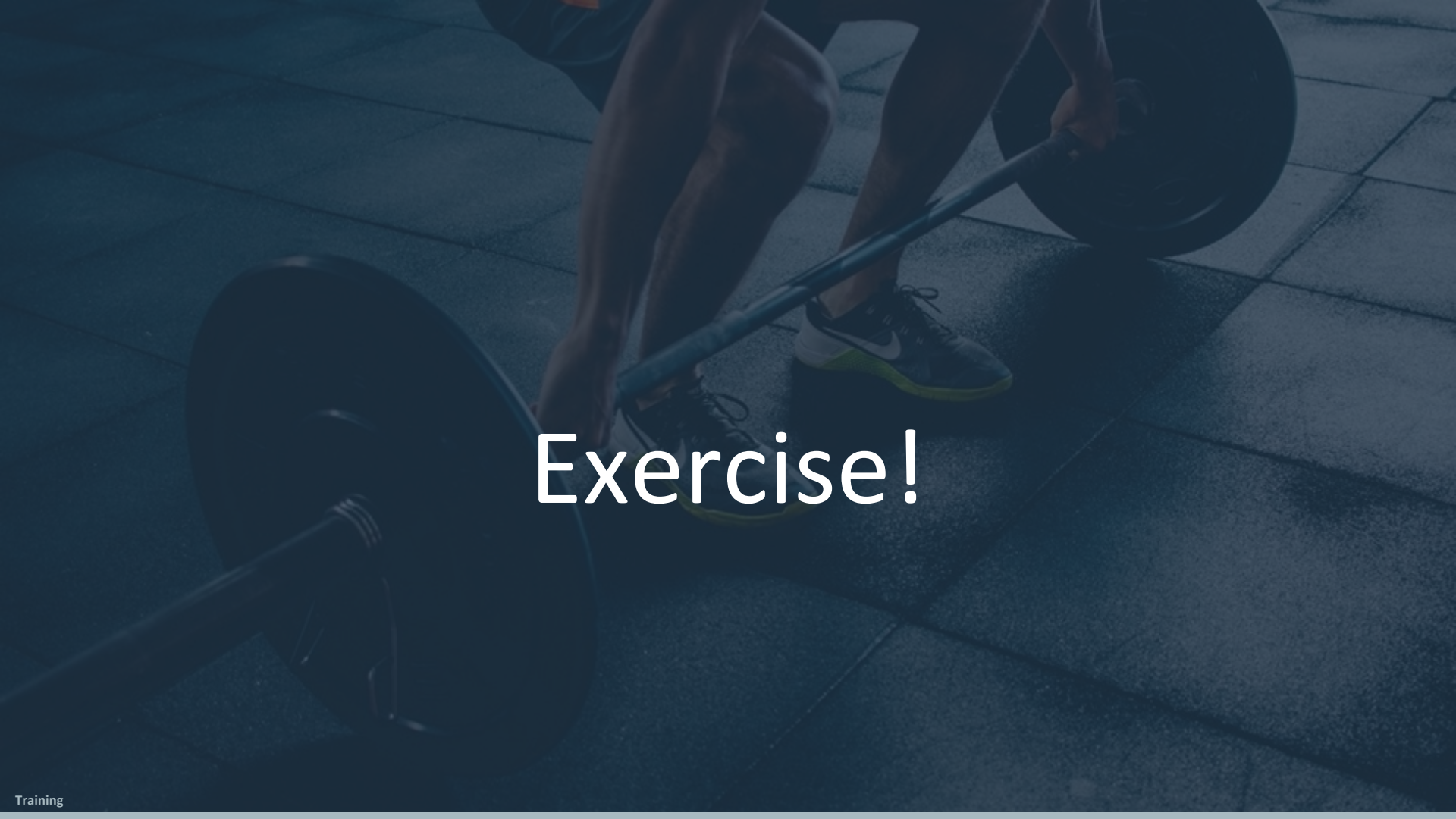
Define in the model, implement in code.

- Similar implementation as service tasks (**Expression**, **Class**, **Delegate expression**)

- **Recommendation**:
  Use expressions and call spring service




```java
@Service
public class ExecutionLoggerBean {

    private static final Logger LOGGER = LoggerFactory.getLogger(ExecutionLoggerBean.class);

    public void logExecutionStart(DelegateExecution delegateExecution) {
        LOGGER.info("Executing {} with field value: {}",
        delegateExecution.getCurrentActivityId(),
        DelegateHelper.getField(delegateExecution, "fieldOne"));
    }
}
```


Exercise!


Exception Handling and Debugging

Different types of Errors

BPMN Errors

Model business exceptions: Paths that deviate from your processes' happy path, but do not constitute a technical error

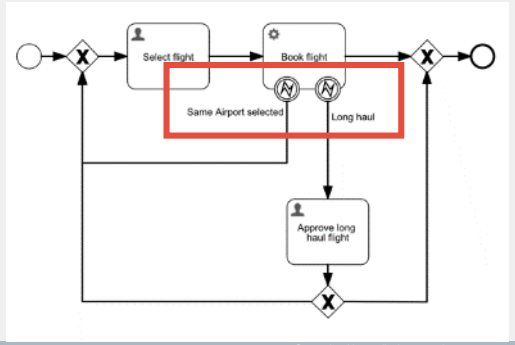
Runtime Errors

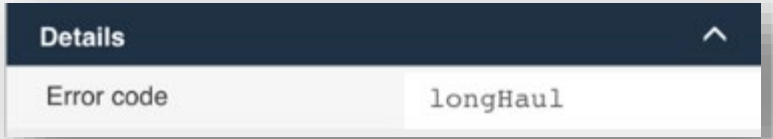
Any unexpected failures that should be mostly of technical nature. Must be handled via the the (REST) API or a tool like Flowable Control

BPMN Error

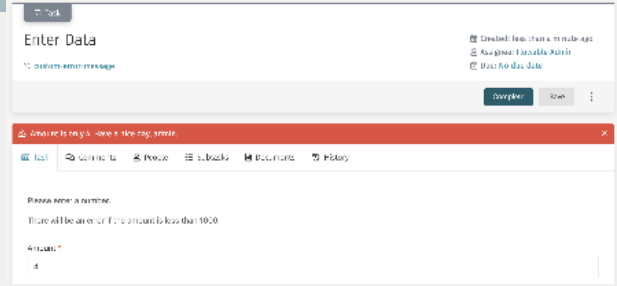
Model your business errors with the Error Event.

- Catch errors with **error boundary events**
- Intended for **business error** – not technical errors.
- Trigger in Java: `throw new BpmnError("longHaul")` or with an **Error End Event**
- Different **error codes** can be caught





Runtime Errors



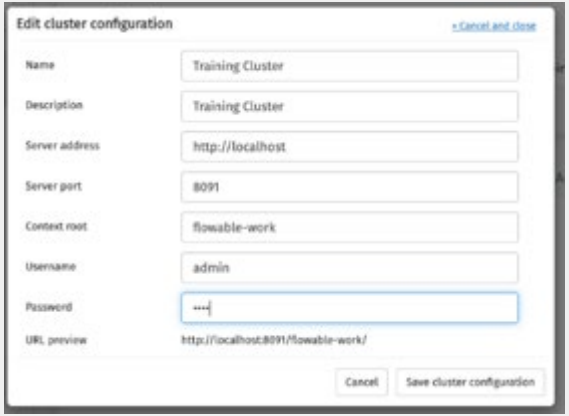
- **Any uncaught Java Exception** will cause service tasks to fail
- If the Service Task is **NOT async**:
  - When completing a human/user task, the user will remain on the form **until the service task is completed**.
  - The REST API will **return a response with the error code 500**.
  - If you throw a FlowableIllegalArgumentException in a JavaDelegate, you can display a **custom error message**. Otherwise, a generic error message is thrown.

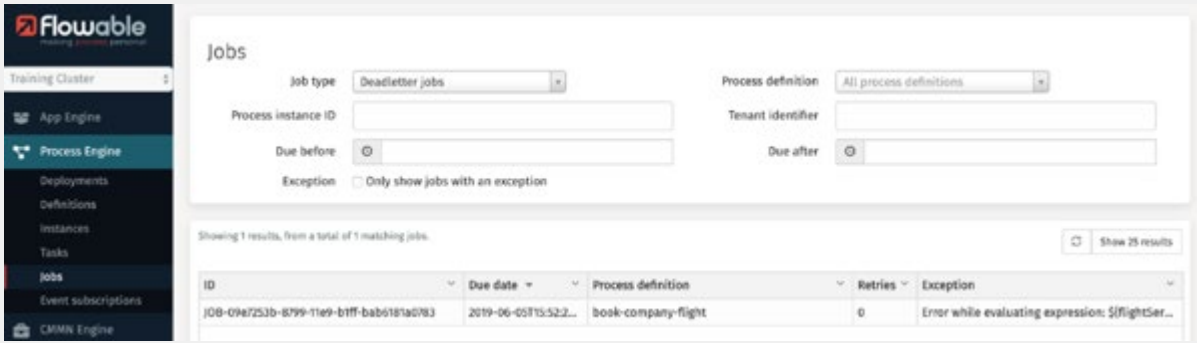
- If the Service Task **IS async**:
  - When completing a human/user task, the task will **immediately be completed successfully,** and the user **will be navigated away from it**.
  - In case of an error, the execution will be reattempted three times*. After that, it is moved to the **dead letter queue**. This behavior can is configurable.
  - From the queue, move the job through Java or REST API or via Flowable Control.

* Can be configured

Flowable Control

1. Configure the instance you want to control
2. View and move dead letter jobs
3. Set variables
4. (And much more)





Debugging

- If you can, **reproduce the issue locally**, ideally with an (integration) test
- If you have access to the DB, inspect the **current** or **historic process** and case state by looking at the respective tables
- With **Flowable Control**, you can debug the process/case state and its variables
- Call /process-api/management/deadletter-jobs/
- If all else fails, turn up the logging levels (`org.flowable`, `com.flowable`, `org.mybatis` for database queries)
- Through spring properties (logging.level…) or at runtime through the actuator endpoint (/actuator/loggers)

Asynchronous Flag

See you later, activity!

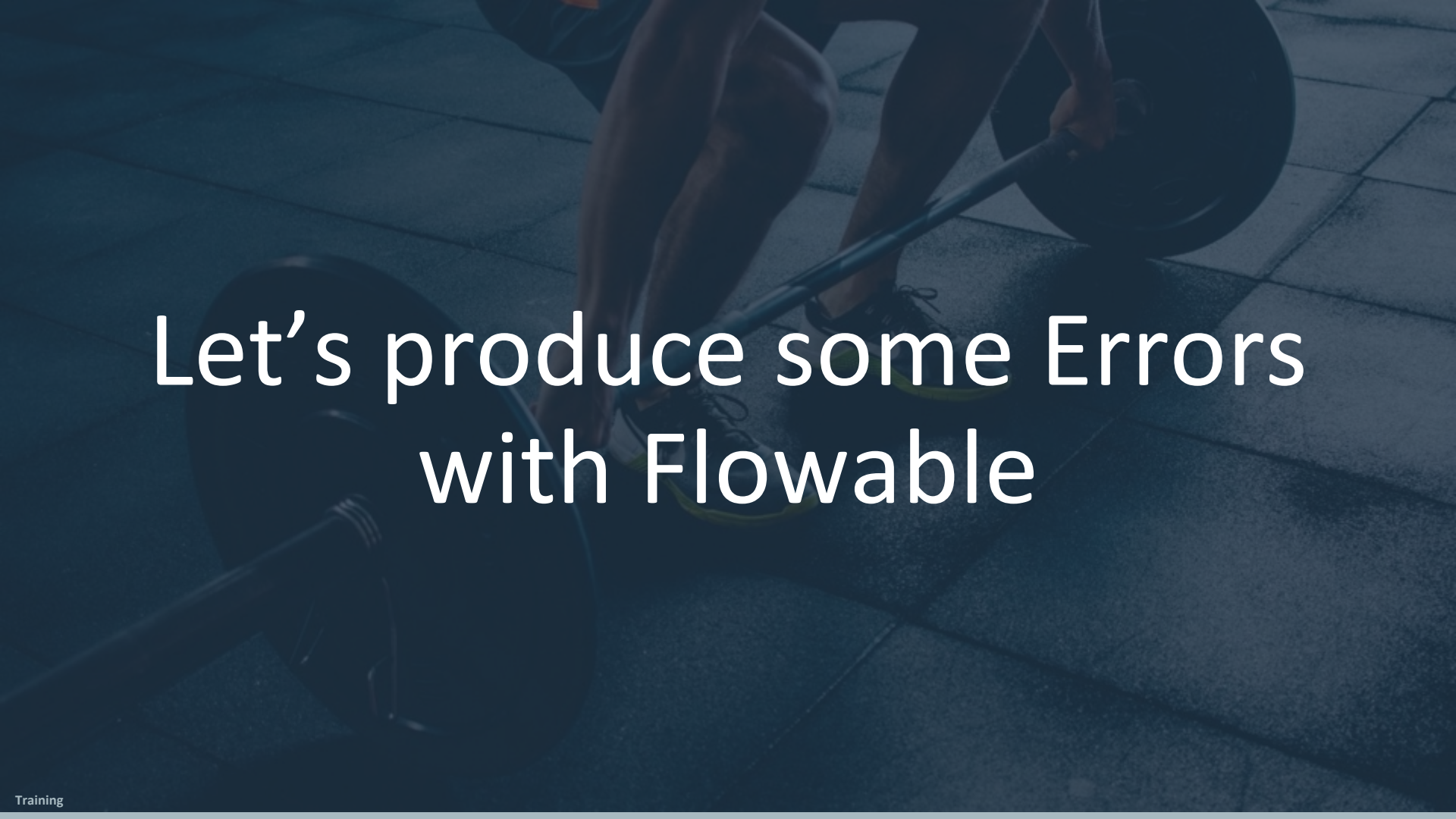
- When an activity (e.g. ServiceTask) is flagged as **asynchronous**, an artificial **wait state** is introduced.
- The job executor performing the previous step **commits** the transaction and creates an **async job** in the database for the next step.
- The **job executor** polls the database for new jobs to perform and creates a new transaction.
- For User tasks followed by long-running transaction, this increases **perceived performance** because it returns control to the user immediately.

Behind the scenes: Async Jobs

- When the execution of a process finds an async ServiceTask, an **async job** is inserted into the ACT_RU_JOB table.
- The async executor has a thread that checks for available jobs and timers.
- When it finds a job, it locks it and adds it to the async executor **queue**.
- The async executor has a thread pool that executes the jobs in the queue. If the queue is full, the job is unlocked and reinserted to the db.

Async Job Errors

- When the execution of a job fails, the async job is transformed to a timer job with a due date.
- It will be picked up later and become an async job again to be retried.
- After a configurable number of failed retries, the job is moved to the `ACT_RU_DEADLETTER_JOB`.
- Dead jobs need to be handled by an administrator to decide the best course of action



Let’s produce some Errors with Flowable

Exercise

Let’s do it the Flowable way.

Goals:

- Flowable Administrator is a technical user. He cannot be greeted. Adapt the model so that whenever he gets greeted, an **error notification task** gets created.
- Use `BpmnError(”code")`
- Greeter and the person to be greeted cannot be the same user. Make sure that a **backend error** is thrown in this case.
- Use `RuntimeException(”error”)`
- Switch between showing a **generic error message** or a **detailed error message** that appears on the UI in this case.
- Use `FlowableIllegalArgumentException(“message”)`.
- Make our greet task **asynchronous**. Try to fix an occurring backend error in **Flowable Control.**

Testing Flowable Applications

Different People, Different Tests

Test to your heart's content.

- BPMN and CMMN are part of your software project: They can be **tested** if needed!
- You can use all **your favorite testing paradigms** and frameworks.
- You can leverage all **Spring Boot Testing** support.
- Here we will concentrate on testing using
  - JUnit 5
  - AssertJ for assertions
  - Mockito for mocking
  - Awaitility to wait for asynchronous events

Test Types

- **Unit Tests**
  - Individual components of a software are tested.
  - Calls to external services are getting mocked.
  - A testing and mocking frameworks are needed.

- **Integration Tests**
  - Multiple components of a software and their interfaces are tested.
  - The DI framework needs to be started.

Integration Tests

- Define a configuration in your **test resources** named `application-test.properties`:

```
spring.datasource.url=jdbc:h2:mem:trainingdb    # Use in-memory DB
flowable.async-executor-activate=false          # Deactivate async executor
flowable.async-history-executor-activate=false  # Deactivate hist. async exec.
```

- Add the following annotations to your integration test class:

```
@SpringBootTest(classes = { YourApplication.class })
@ActiveProfiles("test")
```

  The test is now running as Spring Boot App and uses the default test configuration.
  You can now **inject engines and services** as normal, e.g. through `@Autowired`.

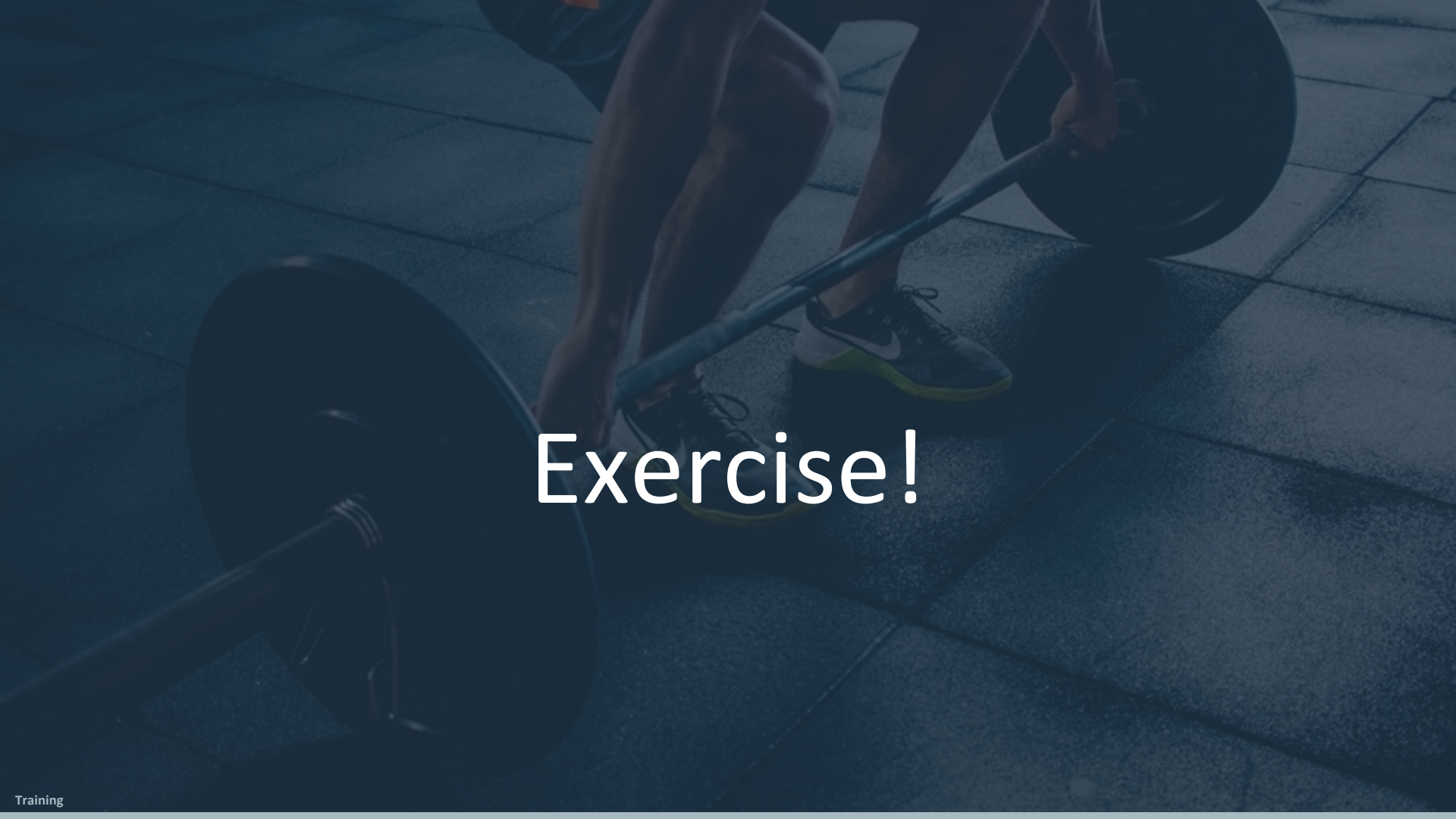
- `TestSecurityUtils` is provided to manage logged in users in the tests (e.g. for checking permissions)

Testing utilities

- You can **deploy any model** through different @Deployment annotations, such as:
  - Apps: `@AppDeployment(resources={”apps/MyApp-bar.zip"})`
  - BPMN: `@Deployment(resources={"processes/test-process.bpmn20.xml"})` available with `@FlowableTest` or `@ExtendWith(FlowableSpringExtension.class)`

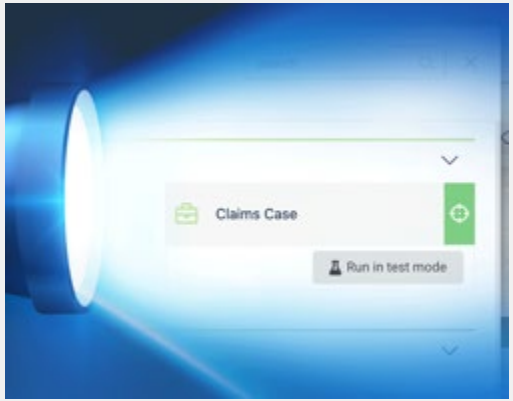
- Unit Test that tests a certain process could therefore look like this:

```
@FlowableTest
public class BpmnUnitTest {
    @Test
    @Deployment(resources = { "processes/test-process.bpmn20.xml" })
    public void myBpmnProcessTest(ProcessEngine processEngine) {
        Mocks.register("syncService", new SyncService());
```


Exercise!

Flowable Inspect

Inspect and Test Cases and Processes



- Increase modeler productivity significantly
  - No more repeated edit-publish-test-fail cycles
  - Follow and modify the execution of cases and processes
- Easily create automated process and case tests
  - Interactively record test scripts by running processes & cases
  - Augment with test actions – validate values and mock services
  - Instantly validate system and service changes
- Business validation of process and case journeys
  - Library providing any number of full or partial runs

Setup

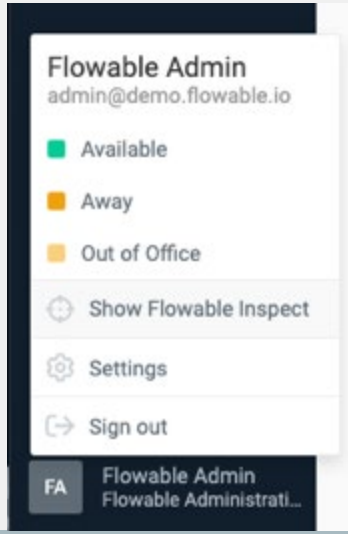
- Add Inspect Dependency to your classpath:

```
<dependency>
    <groupId>com.flowable.inspect</groupId>
    <artifactId>flowable-spring-boot-starter-inspect-rest</artifactId>
</dependency>
```

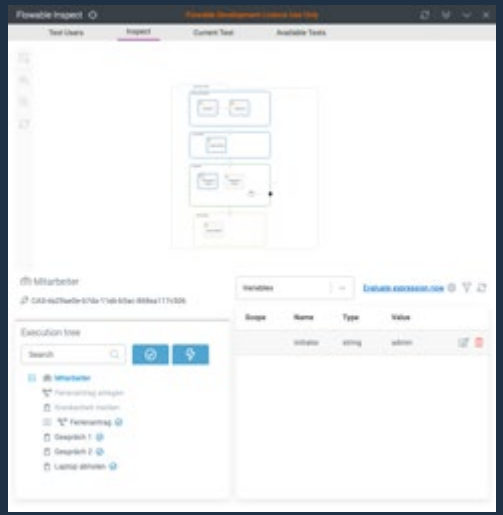
- Enable Flowable Inspect (please don’t do this in production!):

```
flowable.inspect.enabled: true
```

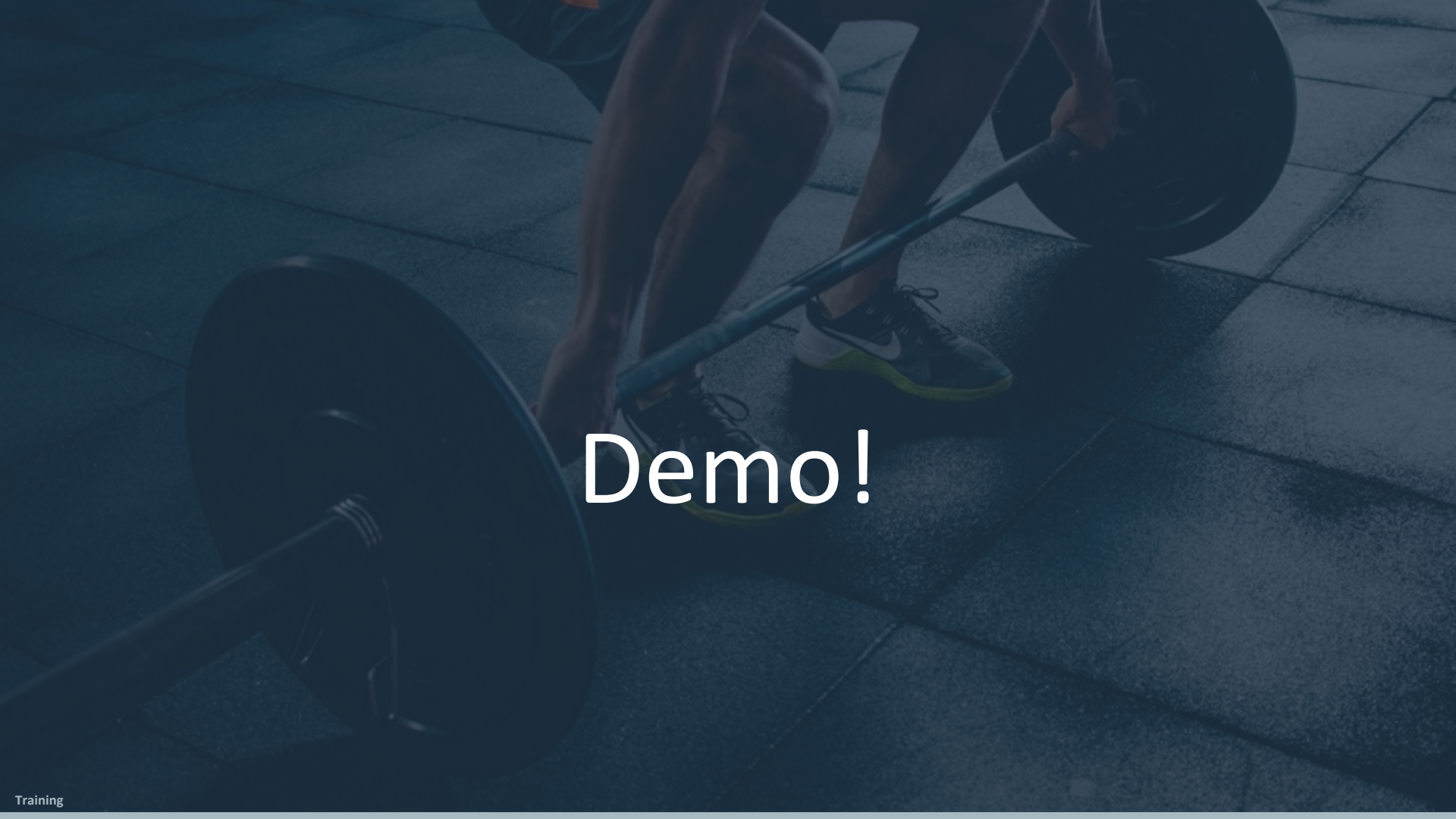
- Click on the profile to enable inspect

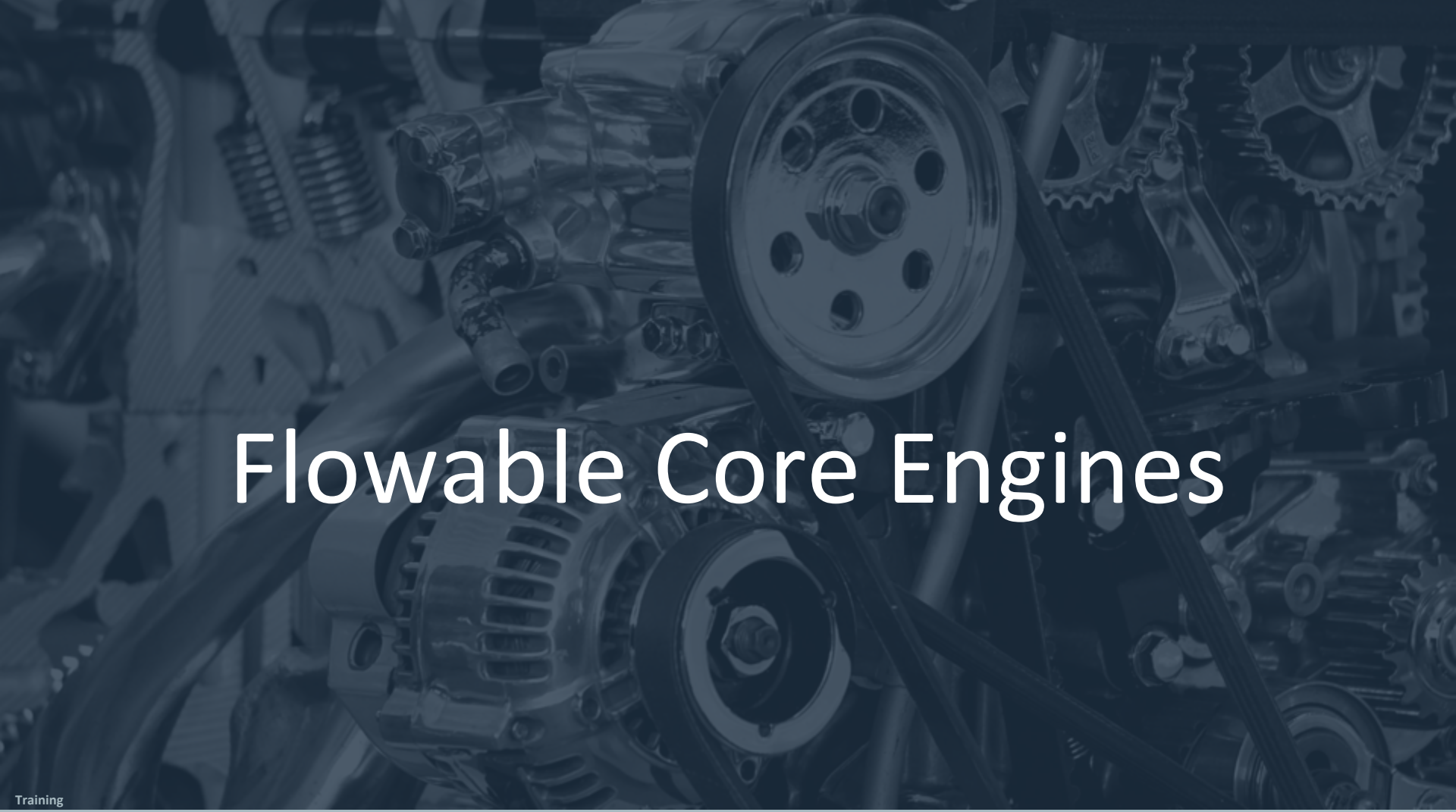


Features



- Test Recording and Execution
- Execution of Assertions
- Breakpoint Usage
- Exception capturing
- Expression evaluation
- Insights into timer, jobs etc.
- Impersonate Test Users (must be enabled)
- ....


Demo!



Flowable Core Engines

Spring Boot

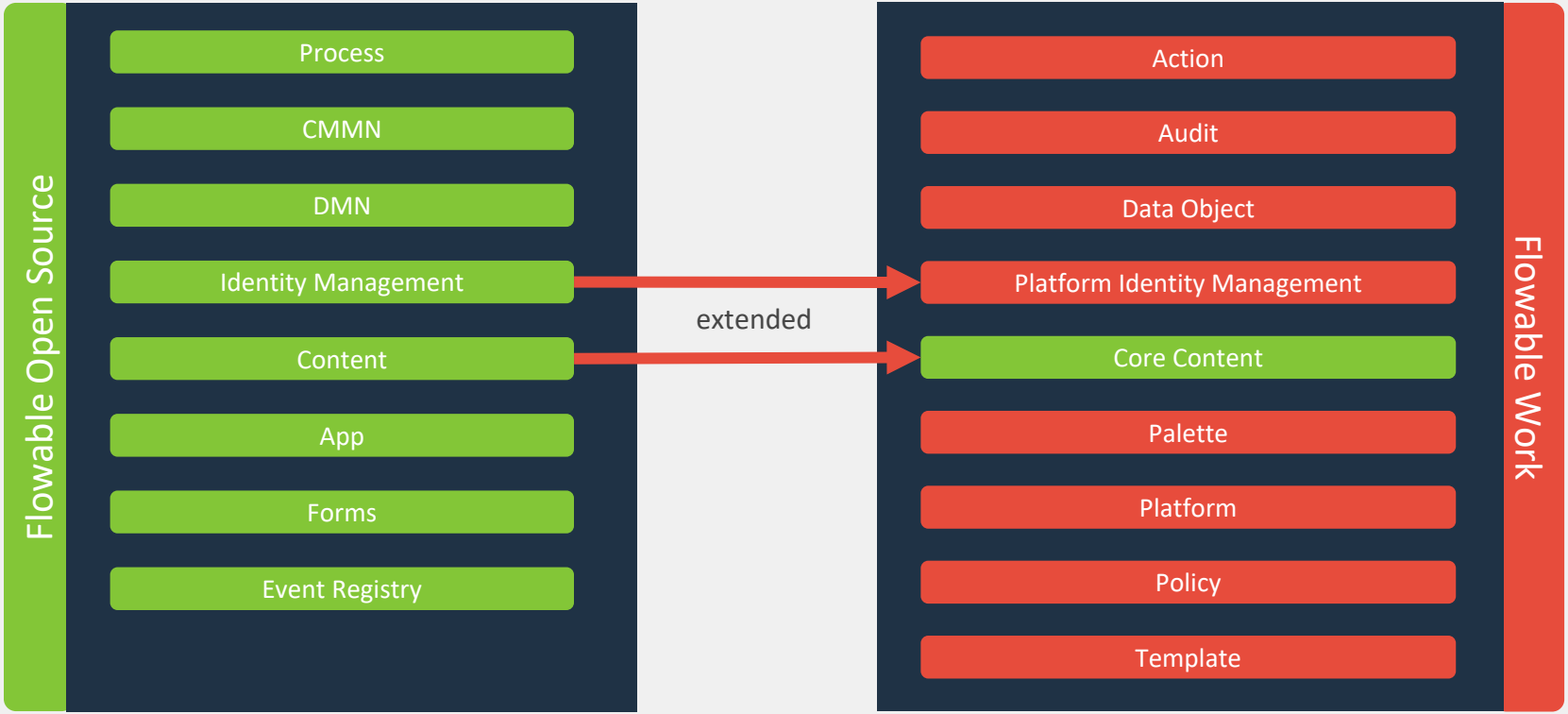
- Flowable is based on Spring Boot
- Start web applications as a simple Java Process (no fat EE application servers).
- Improved maven dependency management with starter dependencies.
- Autoconfiguration based on dependencies.
- Dependency Injection for Service classes
- Defined configuration management with @Configuration classes and application.properties/yml files.

What is an Engine?

Flowable Engines are your lifeblood.

- At the core of most Flowable functionality, there are different **engines**.
- They **encapsulate** certain functionality and **make them accessible** to developers.
- Some things commonly associated with them are:
  - Providing **Java** and **REST APIs**
  - Maintaining **models**, **definitions** and **deployments**
  - Providing **hooks** and **listeners**
  - Low-level operations, e.g. **database, state management** and **Job handling**
- You usually don’t directly interact with the engines but with the **services they expose**!

List of Engines



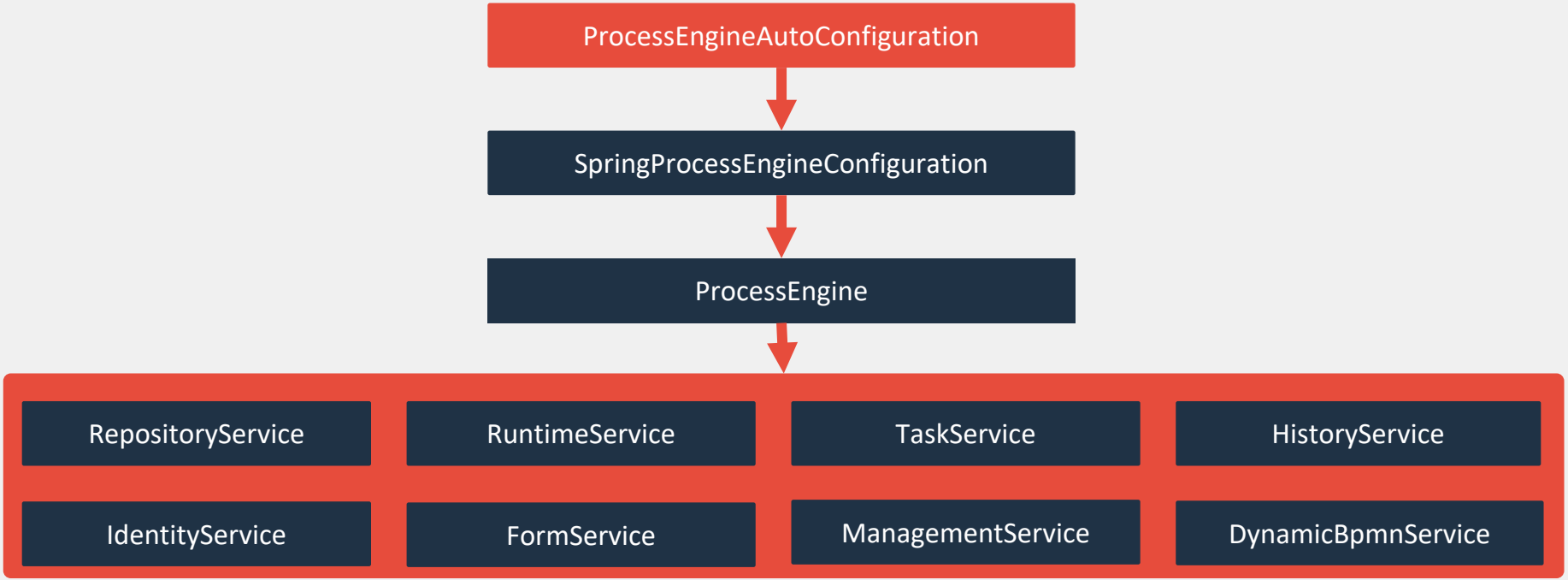
Engine structure

- Engines are created from an **Engine Configuration**

- Each engine provide different services. Examples:
  - **Repository**: manages models, deployments and definitions
  - **Runtime**: starts and controls instances
  - **History**: exposes all data gathered by the engine

- Services offer operations and queries
  - `runtimeService.startProcessInstanceByKey("holidayRequest", variables);`
  - `taskService.createTaskQuery().taskCandidateGroup("managers").list();`

Example: Platform Process Engine



Engine Java API Example 1 / 2

```
// Create the process Engine from a configuration File

ProcessEngine processEngine = ProcessEngineConfiguration
        .createProcessEngineConfigurationFromResource("flowable.bpmn.cfg.xml")
        .buildProcessEngine();

// Obtain the runtimeService

RuntimeService runtimeService = processEngine.getRuntimeService();

// Create a new Process instance

Map<String, Object> variables = new HashMap<>();
variables.put("employee", "John Doe");
variables.put("nrOfHolidays", 5);

ProcessInstance processInstance = runtimeService
        .startProcessInstanceByKey( processDefinitionKey: "P001_Vacation_Request", variables);
```

Engine Java API Example 2 / 2

- History Service:

```
// Retrieve finished activities of a process

List<HistoricActivityInstance> activities =
        historyService.createHistoricActivityInstanceQuery()
                .processInstanceId(processInstance.getId())
                .finished()
                .orderByHistoricActivityInstanceEndTime().asc()
                .list();
```

- Repository Service:

```
// Retrieve available process definitions for a user

List<ProcessDefinition> processDefinitions = repositoryService
        .createProcessDefinitionQuery()
        .startableByUser("userOne")
        .list();
```

Flowable REST Endpoints (1/3)

/process-api/{service}/{operation}/{id}

Example: GET Endpoints

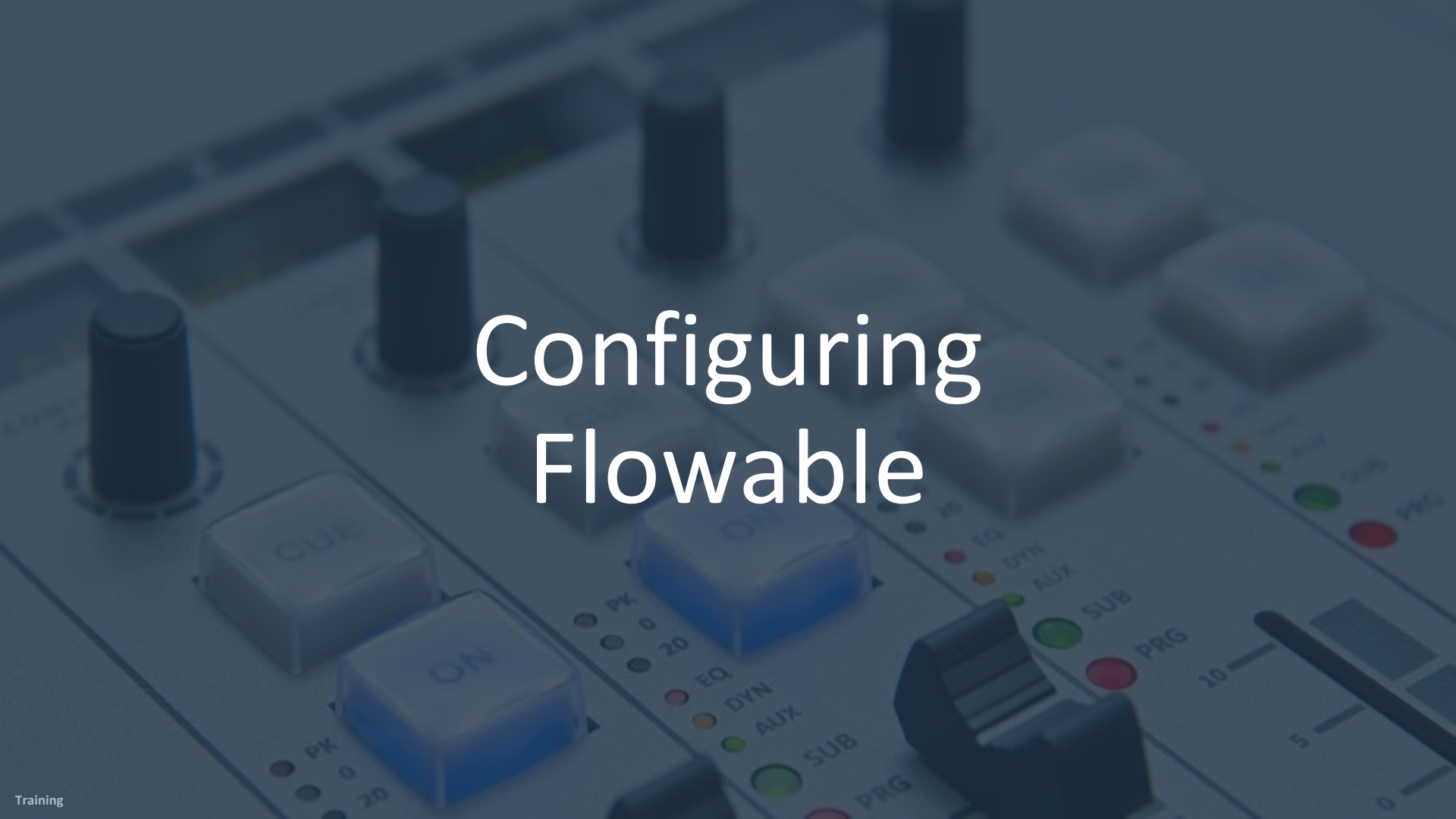
All Instances:	/process-api/runtime/process-instances
Single Instance:	/process-api/runtime/process-instances/{PROCESS_ID}
Definitions:	/process-api/repository/process-definitions
Single Definition:	/process-api/repository/process-definitions/{DEFINITION_ID}
Deployments:	/process-api/repository/deployments
Single Deployment:	/process-api/repository/deployments/{DEPLOYMENT_ID}

Flowable REST Endpoints (2/3)

- The same logic applies to other areas as well, for example:
  - All case instances: /cmmn-api/cmmn-runtime/case-instances
  - Single DMN deployment: /dmn-api/dmn-repository/deployments
  - Single action instance: /action-api/action-runtime/action-instances/{ACTION-ID}
- Find endpoints in code by searching for **`xxxInstanceResource`**, **`xxxInstanceCollectionResource`**, **`xxxDeploymentResource`** etc.
- Swagger documentation is available on documentation.flowable.com
- Paging is available on many endpoints and is the same in open source and platform: Request parameters `start`, `sort`, `order` (asc/desc) and `size`

Flowable REST Endpoints (3/3)

- In general, the endpoints are secured by Spring Security
- Additional security is applied through `RestApiInterceptors`, which are just beans you can override! (e.g.: `CmmnRestApiInterceptor`)
- Fine grained permission control (per App, Process, Task..) is available as **Permission Engine**



Configuring Flowable

Configuration Properties

Configure Flowable the “Spring Boot Way”

- Flowable applications are based on **Spring Boot**, so you can configure most things in an `application.properties/yaml` file.
- All **Flowable properties** start with ”`flowable.`”
- Available configuration properties and values, can be retrieved through the following endpoint if activated: `/actuator/configprops`.
- You can find a reference for all properties in the **online documentation**.

Common Configuration Properties

Datasource configuration:

- `spring.datasource.driver-class-name=org.h2.Driver`
- `spring.datasource.url=jdbc:h2:mem:my-db;`

Model Autodeployment prefixes and suffixes (in ANT syntax)

- `flowable.process-definition-location-prefix=classpath*:/processes/`
- `flowable.process-definition-location-suffixes=**.bpmn20.xml,**.bpmn`

Mail properties

- `flowable.mail.server.host=localhost:39831/my-mail-server`

Configuration approaches

- There are several ways to configure Flowable applications, but some of them are easier/better. YMMV, anyway. Summarizing:
  - Engine **Configuration Configurers**
    - This is the most simple and effective way to configure the Flowable engines after the aforementioned configuration properties.
    - The ”Configurer” classes provide an interface to do almost everything with the engines.
    - There is a long list of “Configuration Configurers”, one for each engine.
  - **Custom EngineConfigurator**
    - If you need to do configure the engine right after the configuration is initialized.
    - Just for very special cases.

Configuration Configurers

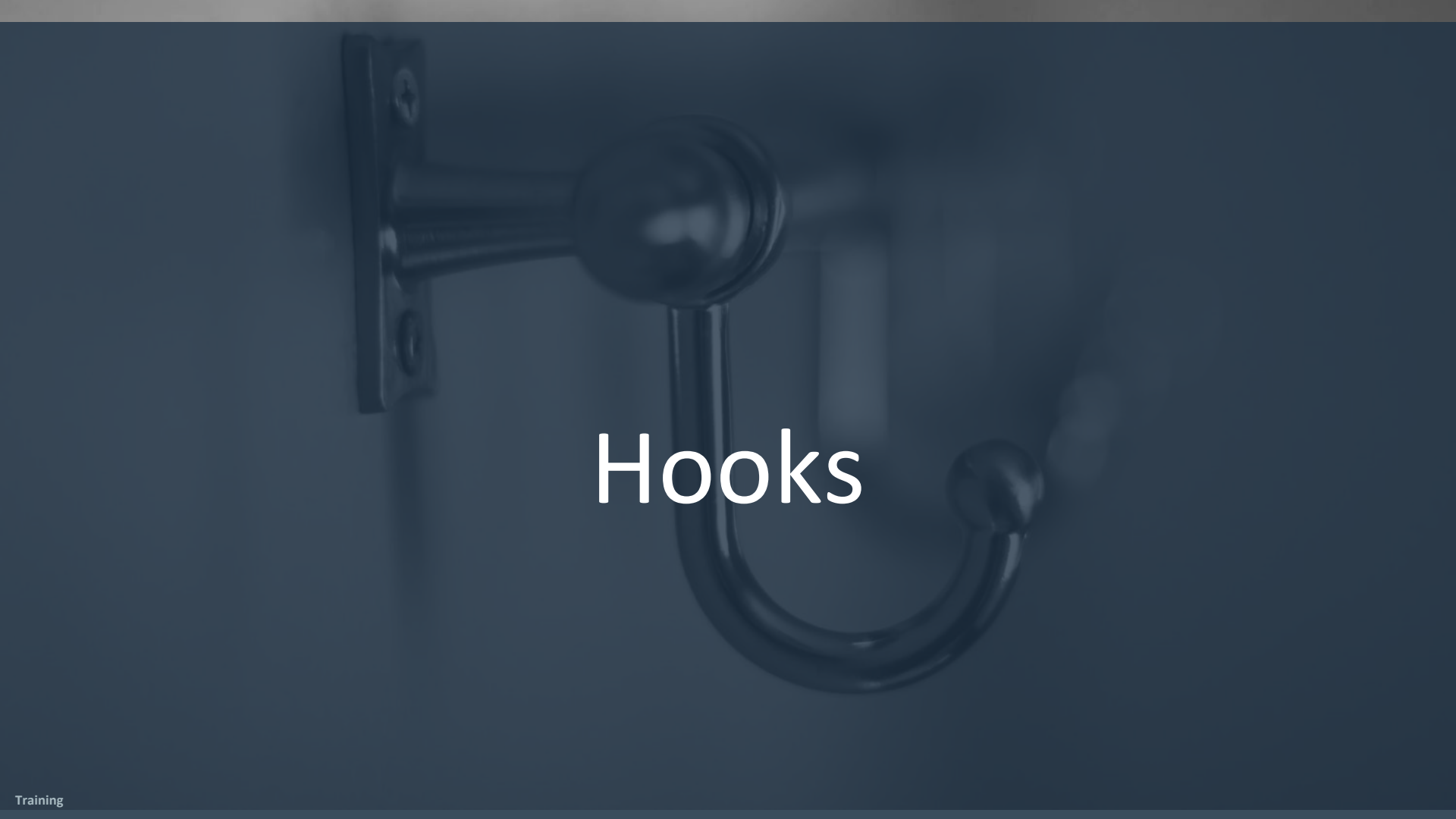
OSS	SpringProcessEngineConfiguration
	SpringCmmnEngineConfiguration
	SpringDmnEngineConfiguration
Core	AppEngineConfiguration
	SpringContentEngineConfiguration
	SpringFormEngineConfiguration
	CoreIdmEngineConfiguration
Platform	ActionEngineConfiguration
	AuditEngineConfiguration
	DataObjectEngineConfiguration
	TemplateEngineConfiguration
Engage	EngageEngineConfiguration

Configuration Configurer Example

```
@Configuration
public class CustomProcessEngineConfig  {

    @Value("${http.connection.connectTimeout:5000}")
    private int connectTimeout;

    @Value("${http.connection.socketTimeout:5000}")
    private int socketTimeout;

    @Value("${http.connection.connectionRequestTimeout:5000}")
    private int connectionRequestTimeout;

    @Value("${http.connection.requestRetryLimit:3}")
    private int requestRetryLimit;

    @Bean
    public EngineConfigurationConfigurer<SpringProcessEngineConfiguration> customProcessEngineConfigurer() {
        return engineConfiguration -> {
            engineConfiguration.setEventListeners(ImmutableList.of(new CustomListener()));

            HttpClientConfig client = new HttpClientConfig();
            client.setConnectTimeout(connectTimeout);
            client.setSocketTimeout(socketTimeout);
            client.setConnectionRequestTimeout(connectionRequestTimeout);
            client.setRequestRetryLimit(requestRetryLimit);
            engineConfiguration.setHttpClientConfig(client);
        };
    }

}
```

Custom Listener

Custom HTTP Client Configuration


Hooks

Hooks

Define in the model, implement in code.

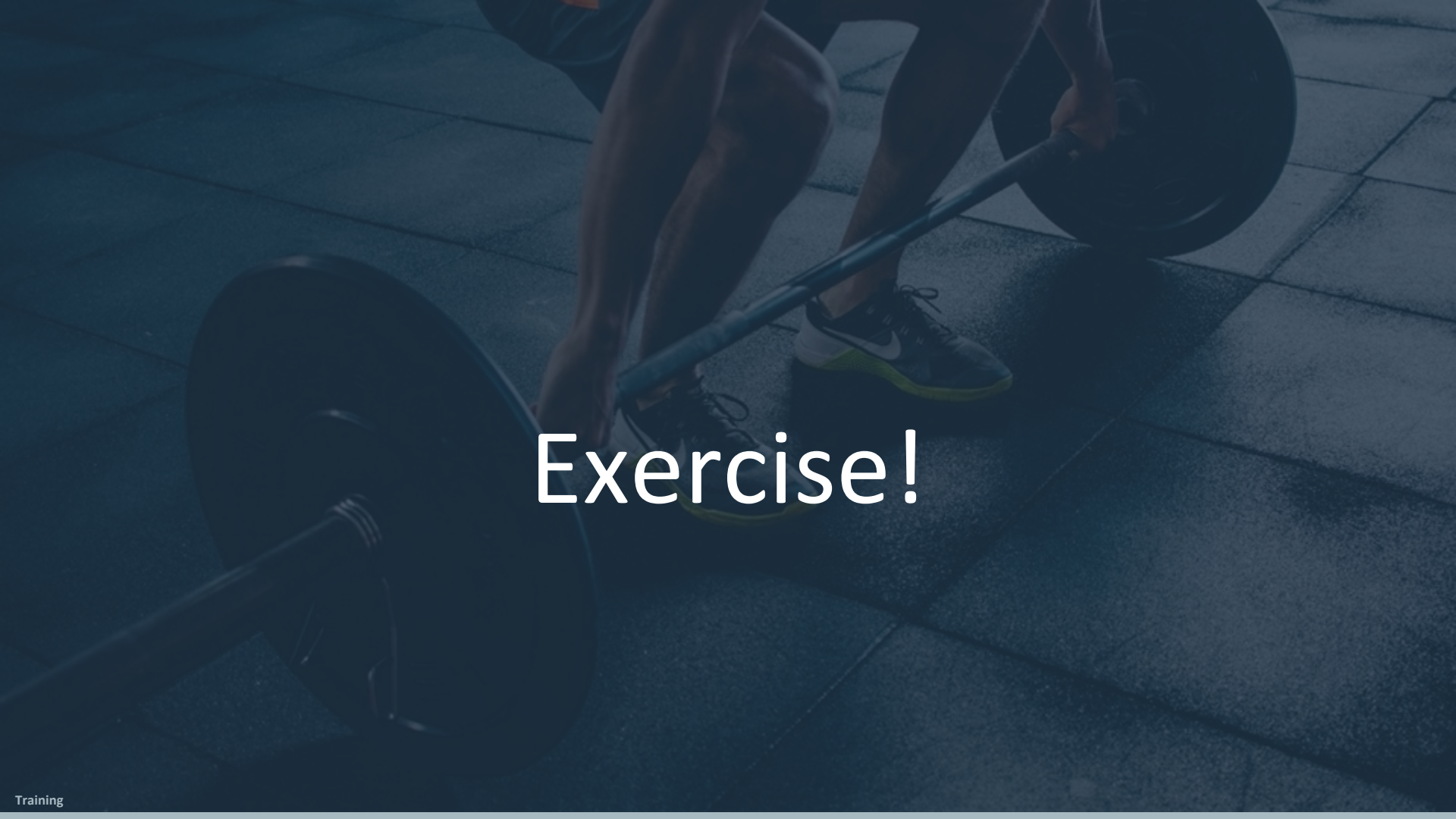
- The engine configurations provide powerful ways to extend them with **custom hook functionality**

- Motivations:
  - Execute code on certain events **independent of a specific model.**
  - e.g. add custom logging

- Most important ones:
  - Event Listeners – Like model listeners, but programmatically
  - Parse Handlers – Hook into the parse process of the BPMN or CMMN file to add features to specific activities

Hooks Implementation

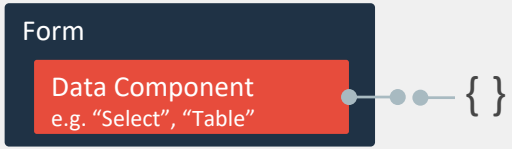
- Register a EngineConfigurationConfigurer for an Engine
  - E.g.:
    ```
    @Configuration
    public class BpmnListenersConfiguration implements
    EngineConfigurationConfigurer<SpringProcessEngineConfiguration>
    ```
  - Add listeners or parse handlers
    ```
    engineConfiguration.setPostBpmnParseHandlers(postBpmnParseHandlers);
    engineConfiguration.setEventListeners(eventListeners);
    ```
  - Listeners inherit from `AbstractFlowableEventListener`


Exercise!

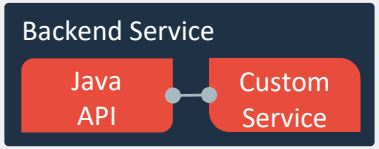

Break


Integrating
Flowable
Applications

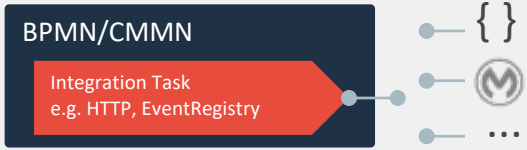
Integration Patterns



- Fetch data from a REST API
- State change only in front end

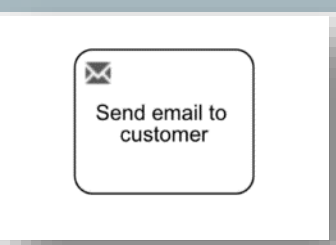


- Develop custom service and logic
- Compile against public Java API
- Load in Engine classpath



- Call external system from process
- Bind data from endpoint response
- Send data to endpoint

- Develop independent microservice
- Integrate via the Public REST API

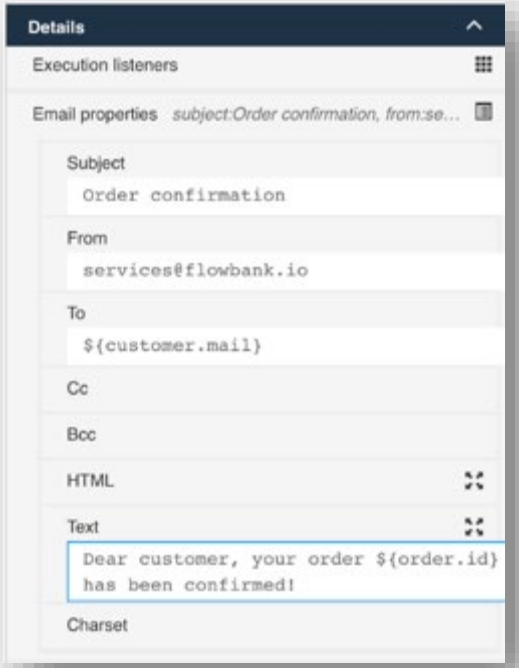


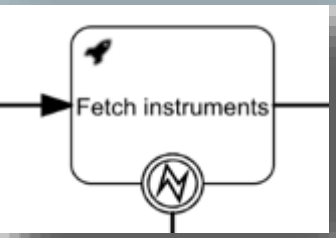
Mail Task

- Send Emails with the Mail Task **directly from CMMN and BPMN**

- Configure once in your properties:

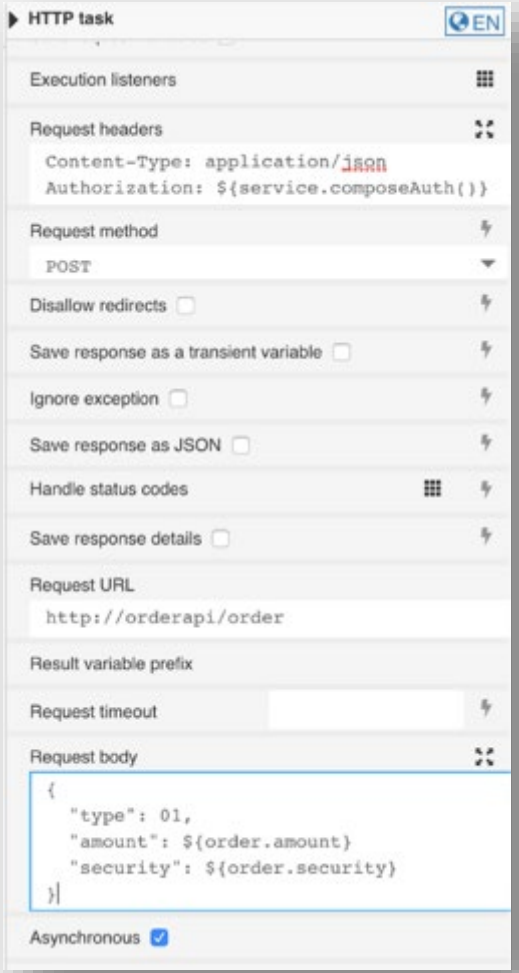
```
flowable.mail.server.username=yourUser
flowable.mail.server.password=yourPassword
flowable.mail.server.use-ssl=false
flowable.mail.server.use-tls=false
flowable.mail.server.port=1025
flowable.mail.server.host=localhost
flowable.mail.server.default-from=flowable@localhost
flowable.mail.server.force-to=test@flowable.com
```





HTTP Task

- **No-code** execution of HTTP calls
- Compose both headers and body dynamically with **expressions** for maximum flexibility
- Map the response body to **case** or **process variables**
- Simple error handling based on **HTTP status codes**



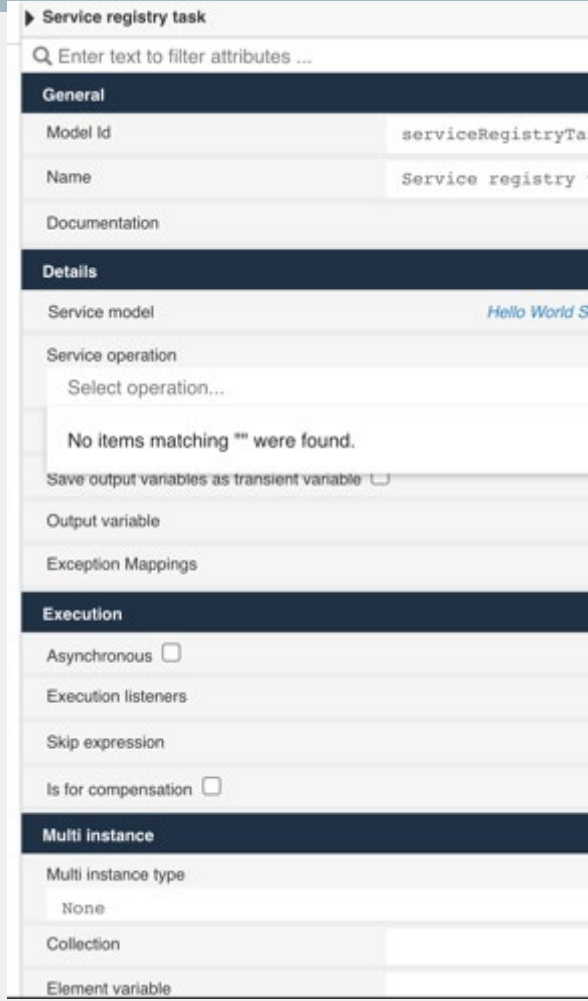
General Architecture Rule

- For **synchronous** communication with APIs use the Flowable **Service Registry**.

- For **ansynchronous** communication with APIs use a messaging system and the Flowable **Event Registry.**

Service Registry

- **No-code** call of external REST APIs
- Configurable **Operations and Input/Output values**
- Resuable with Service **Registry Tasks**
- Mapping of to **case** or **process variables**
- Support for authentication
- Support for Expressions, REST APIs and Data Objects



Service Registry

General configuration

A **standard** service model defines operations with input and o
and managed. This kind of service model has less freedom to

Type: Standard

Service type: REST

REST Settings

Base URL

Custom headers: + Add header

Operations

+ Add operation

New operation

Label: Operation 1

Key: operation1

Description

REST Configuration

URL

Method

Headers: + Add header

Output path

Input parameters

Input parameters define what type of data this operation expects. When modeling a BPMN or CMMN model, runtime variables are mapped to these parameters. These parameters are available when configuring the operation (for example, as a value in an expression or as a field for a REST body).

Add parameter

Label: Input 1

Name: input1

Type: String

Body location

Excluded from body

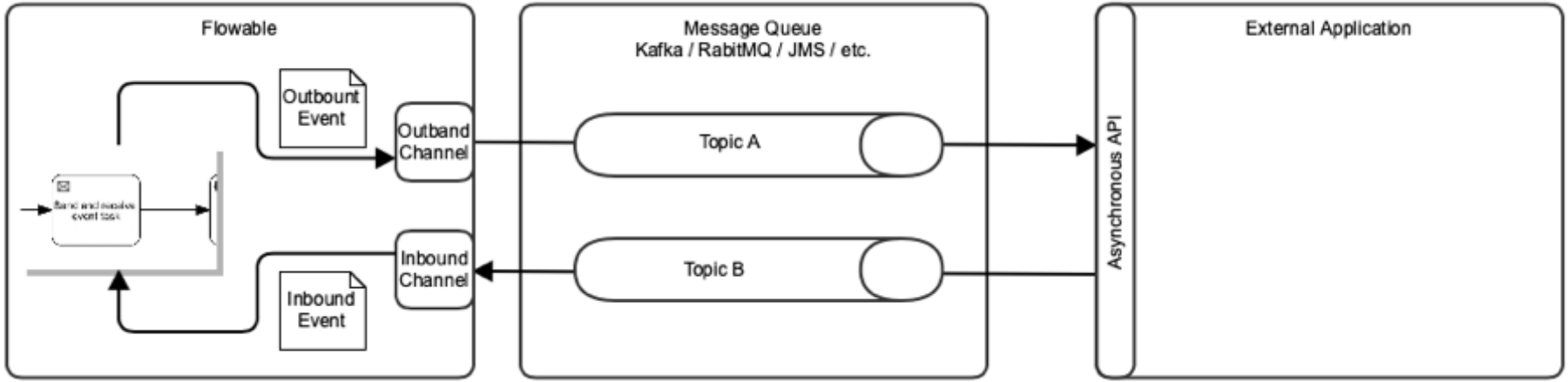
Description

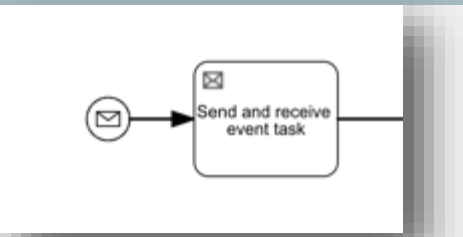
Required

Default value

Add | Cancel

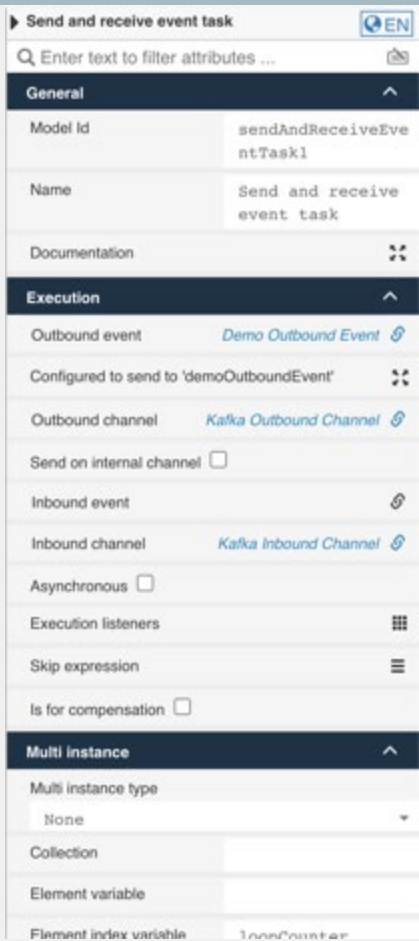
Asynchronous Communication



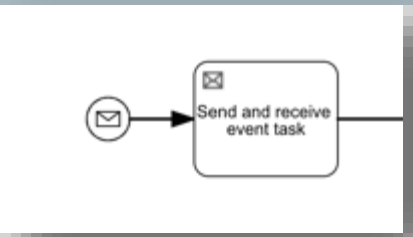


Event Registry

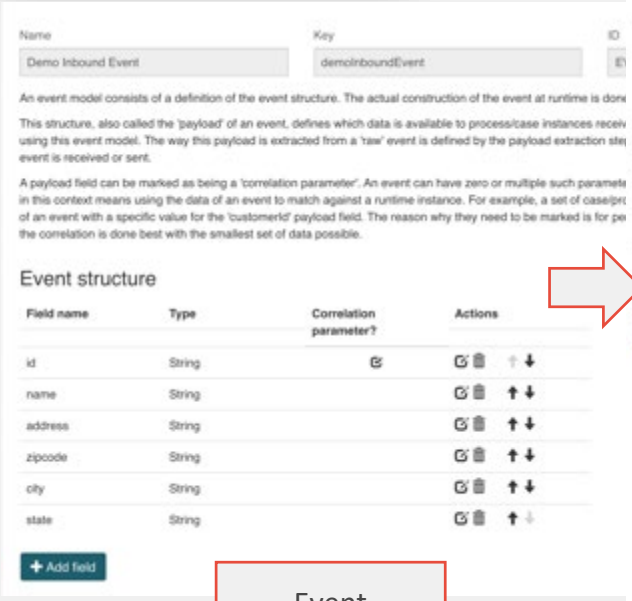
- **No-code** execution of Messages
- Configurable **Channels and Events**
- Integration with **common message providers and receive emails**
- Mapping of event to **case** or **process variables**
- **Correlation** of variables with event parameters
- **Start** processes and cases and send/handle **intermediate** events



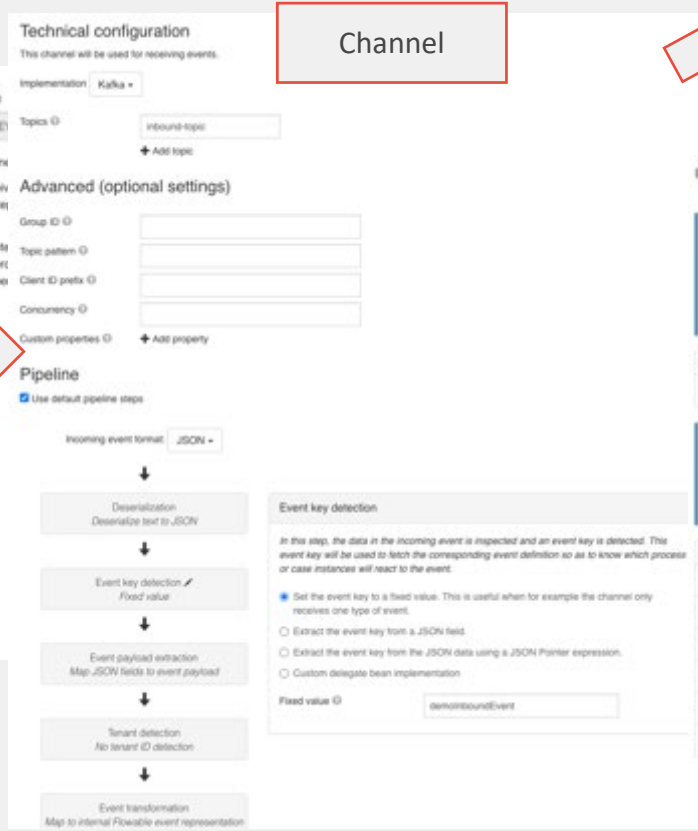
Event Registry

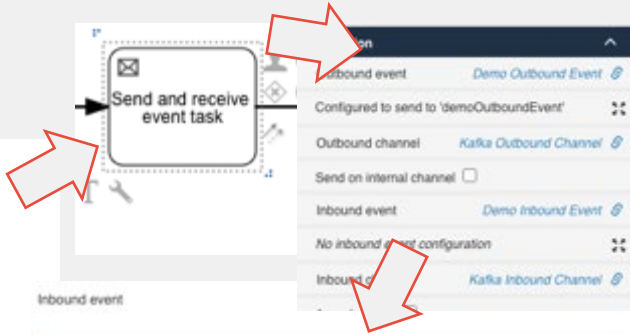


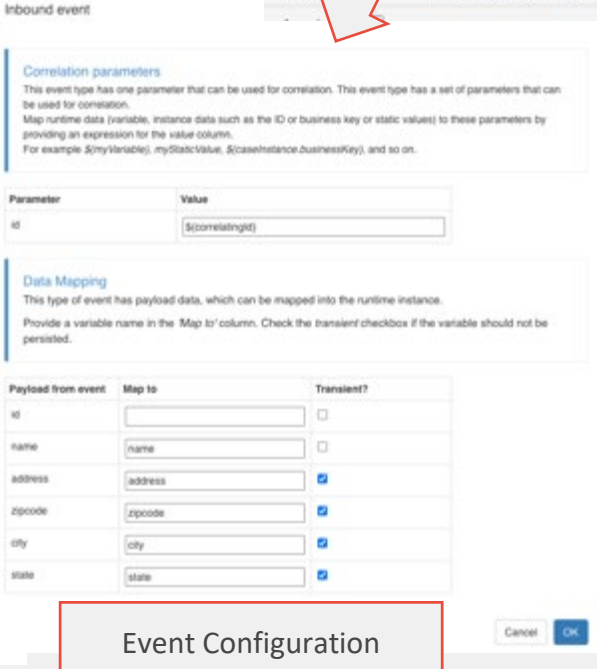
Channel



Event





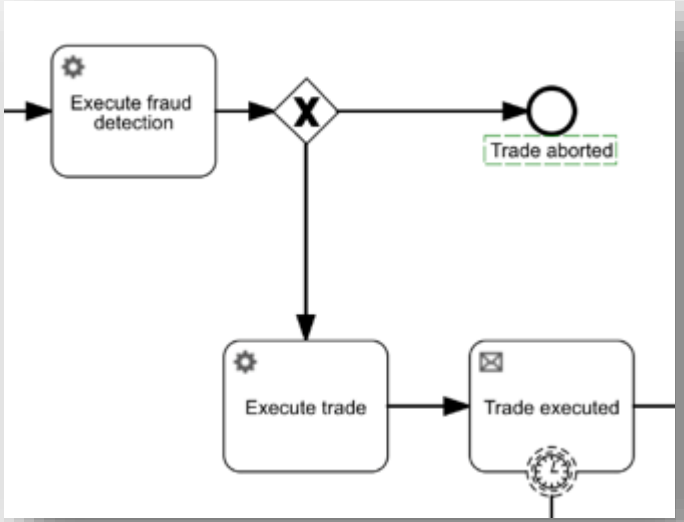


Event Configuration

Triggerable Service Task

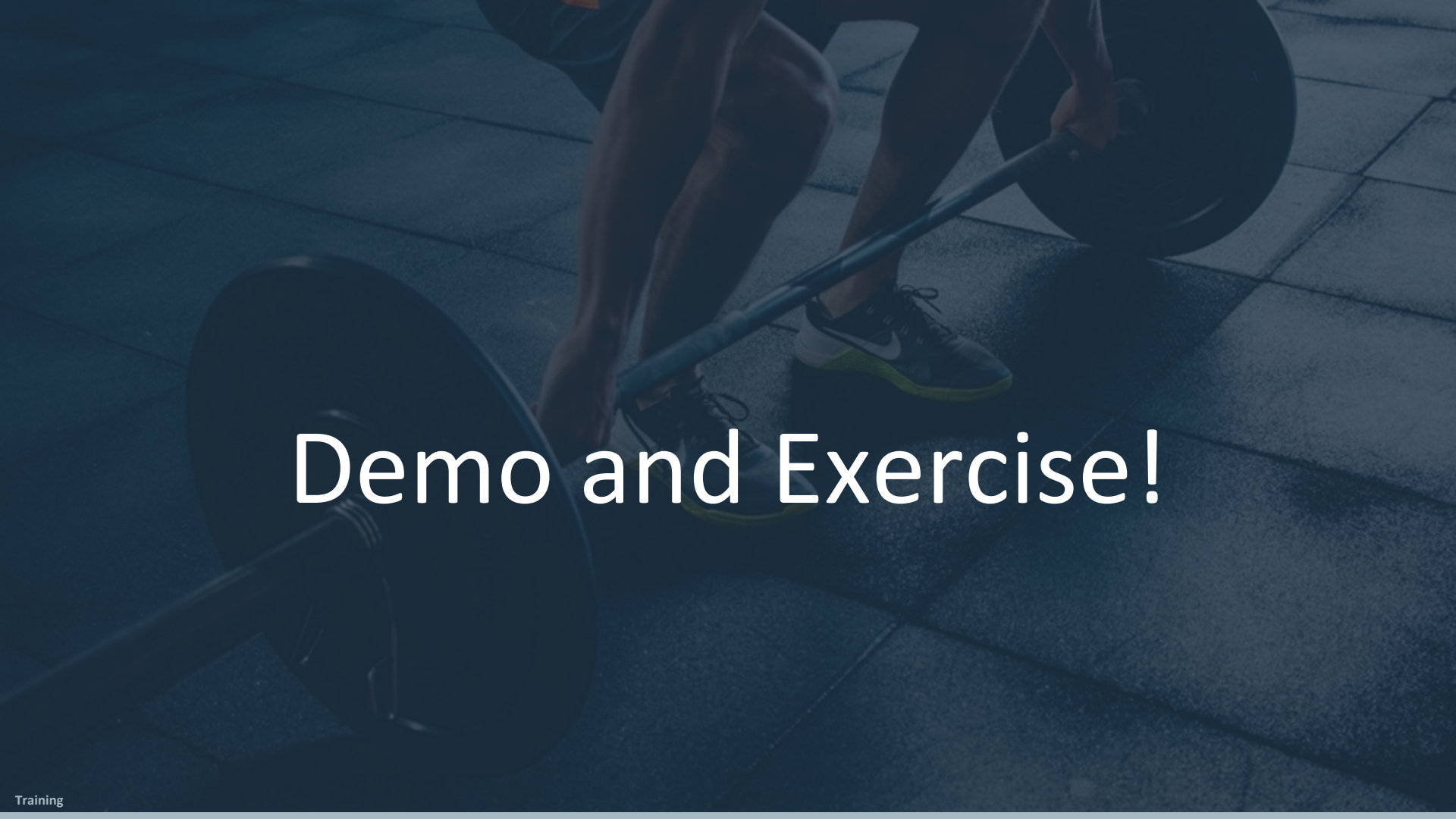
Use the ‘triggerable’ property on the service task for potentially **Long Running Service Tasks.**

Otherwise, your service tasks will block the job execution. If the job takes longer than the configured transaction duration (default: five minutes), the job will fail



Triggerable ☑

```
@Service
public class FraudDetectionService implements
TriggerableActivityBehavior, JavaDelegate {
```


Demo and Exercise!

Operations

Running Flowable Platform in different Scenarios

Architecture

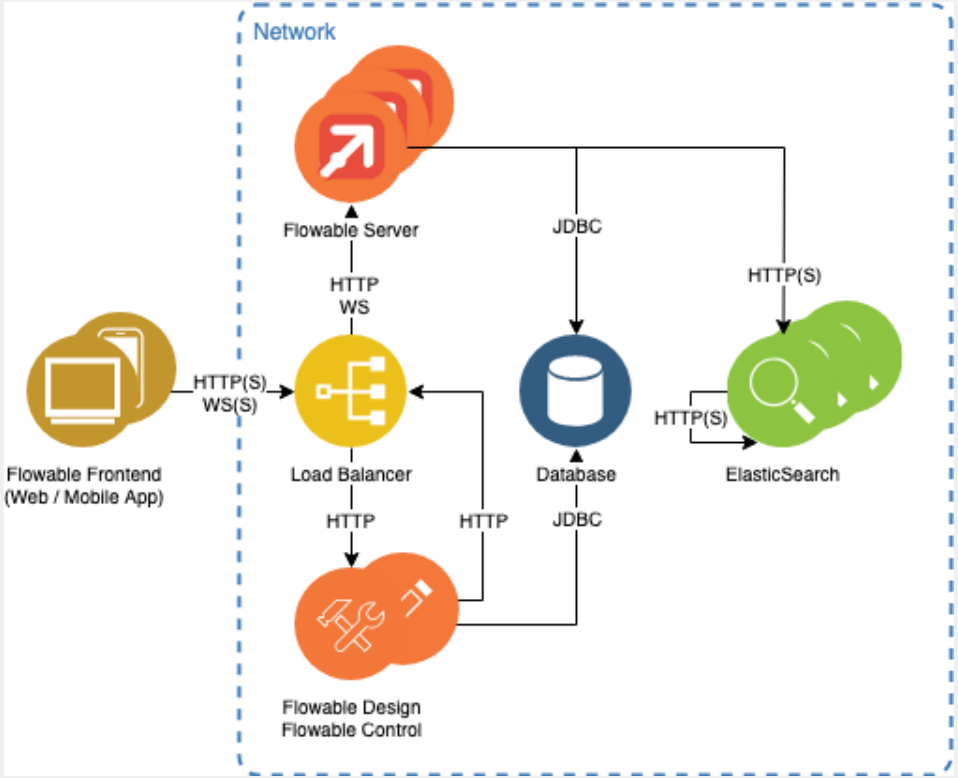
Flowable Server

- Based on Java and Spring Boot

Flowable Frontend

- Based on React, can be served by Flowable Server or any HTTP Server

WebSocket communication is required for Flowable Engage



Prerequisites

Flowable is based on Java and Spring Boot

For a complete Overview of supported Versions, see docs.flowable.io

Servlet Container

- Any Servlet >= 3.1 container
  and the embedded containers supported in Spring Boot
- Tomcat 9.X is recommended

Database

- Oracle, PostgreSQL, MySQL, MSSQL and DB2 are supported

Elasticsearch

- Required for indexing features (mandatory for Flowable Work/Engage)
- Should be run as a cluster in production

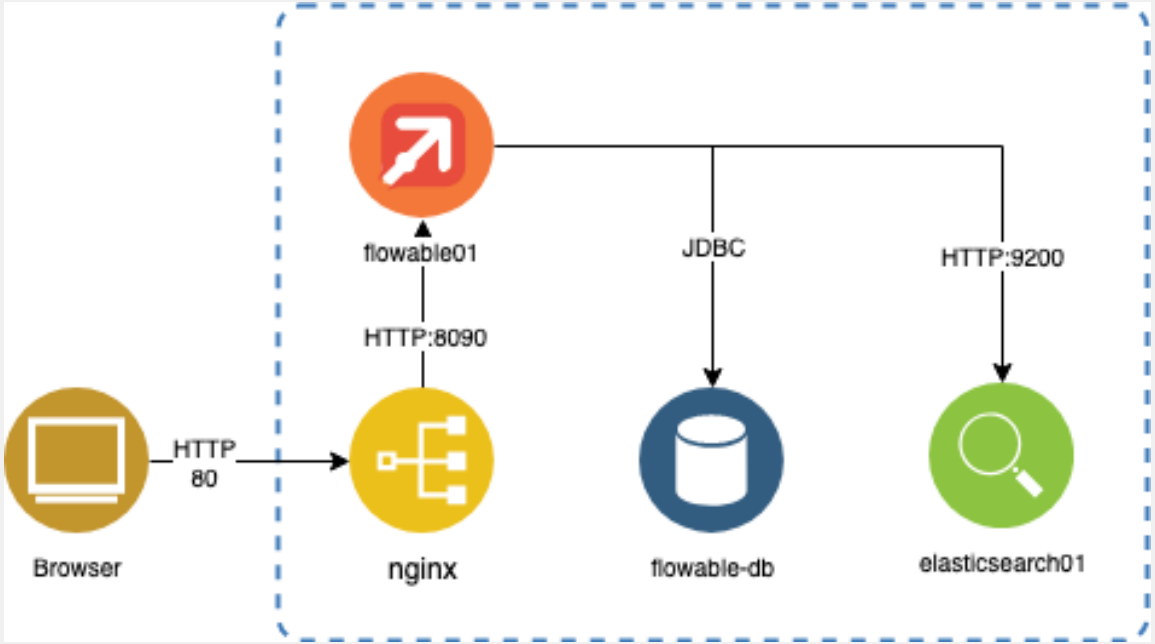
Browser

- Chrome, Firefox, Internet Explorer and Safari are supported

Sizing

- Sizing of Servers, DB and ElasticSearch depends on the project
- However, some guidelines can be found in the installation guide under docs.flowable.io

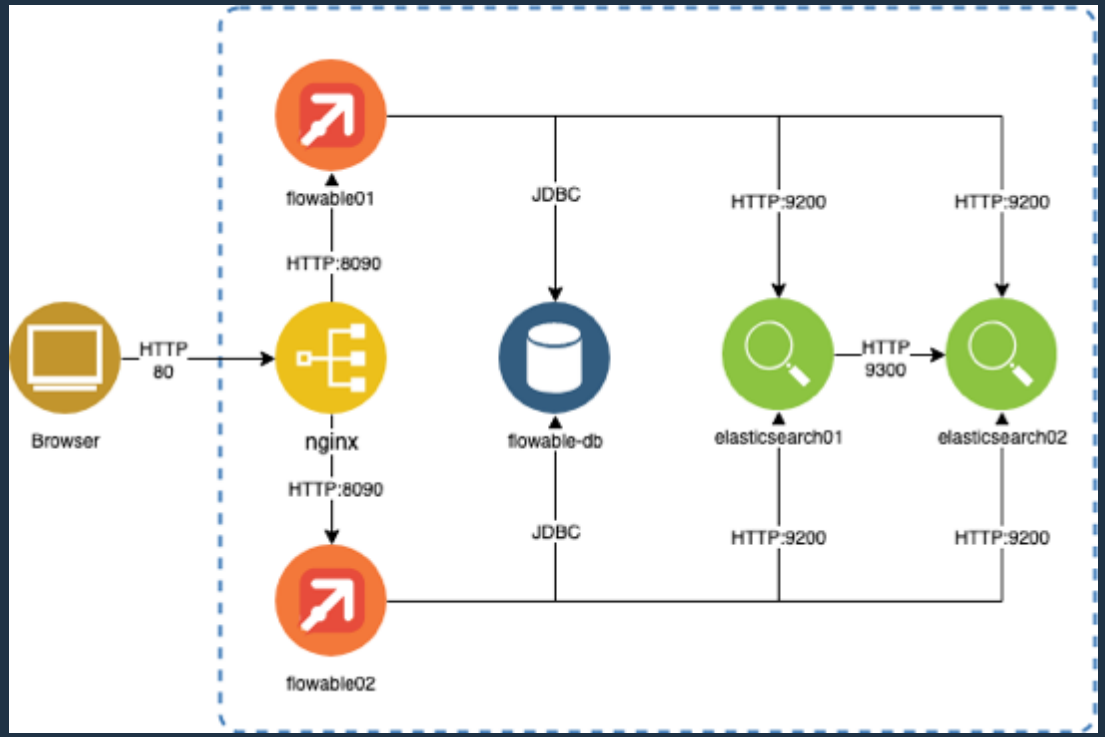
Running a single node



Running clustered

Flowable keeps all its state in the database, therefore **no additional synchronization is required**

Several instances work on the same DB by using **optimistic locking**. Every instance can be configured to fetch jobs. They are locked during execution in each instance



Development setup

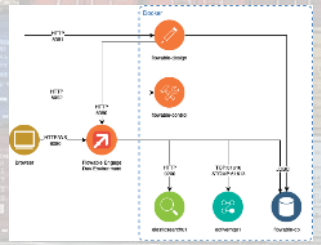
- We recommend installing **Elasticsearch**, a **database** and ActiveMQ (for Engage) locally
- Alternatively, you can **run** them **with Docker**

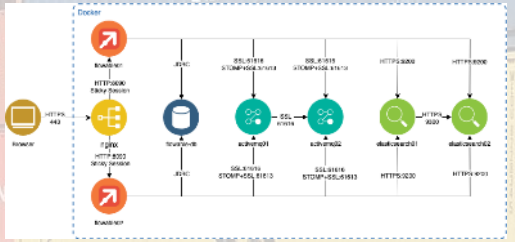
- Run Tomcat locally to deploy Design and Control, or as Docker Container. Create a custom project in case customizations required

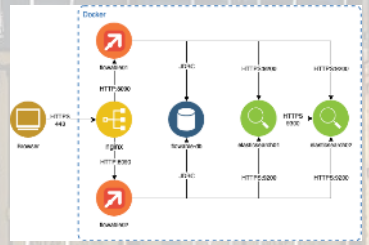
- See Installation

Using Docker

- Flowable provides **docker images** for Design, Control and even Work/Engage
- Unless you do no customization at all, you will **build your own image** at least for Work/Engage
- Flowable runs in any cloud environment







Backup and Restore

- Mandatory Backups:
  - Database
  - Content Storage (e.g. Filesystem)

- Optional Backups:
  - Elasticsearch: Index can be rebuilt with DB

Detailed guide:

https://docs.flowable.io/latest/admin/tech-notes/backup/

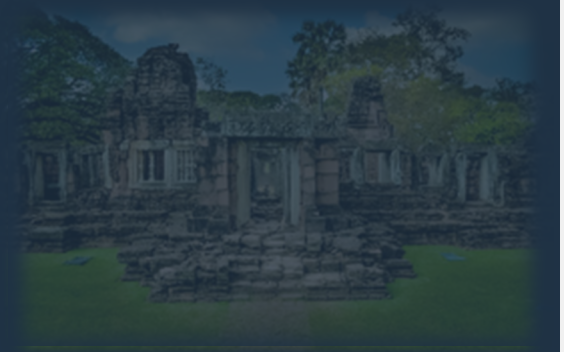

Monitoring

There are many ways to monitor Flowable

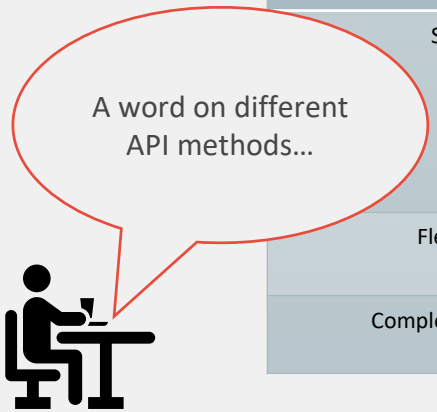
- History
- Audit Engine
- Jobs
- Actuators
- Logging
- Pro-active Alerting

History

Use it for business related monitoring (e.g. KPIs, business alerting…)



- Using the engine respective history can be used to monitor everything about the state of the engines
- Can be accessed via Java API, DB or REST



	Java	DB	REST
Stability	Changes result in compile error	Changes result in runtime error	Changes can be checked against SWAGGER definition or result in runtime error
Flexibility	Can combine calls in Java	Can use any SQL features	Can use only what is available
Completeness	Complete	Complete + internals	Simplified

Audit Engine

Use it for fine grained business events

- Add audit events by using the audit engine
- Query them via REST, Java or DB

Job tables

Use it to get alerted on job failures

- Again, Java, REST or DB can be used, but REST is sufficient here, e.g.:

1. /process-api/management/jobs?messagesOnly=true
2. /process-api/management/jobs/{jobId}/exception-stacktrace

Actuators

Standard Spring Boot feature extended by Flowable

Other endpoints provide http trace, log files, all controller mappings and more. See:

https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html

- Health endpoint delivers information about application health, e.g.:

- GET /actuator/health


```
{
  "status": "UP",
  "details": {
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 101241290752,
        "free": 88578781184,
        "threshold": 10485760
      }
    },
    "db": {
      "status": "UP",
      "details": {
        "database": "PostgreSQL",
        "hello": 1
      }
    },
    "elasticsearchRest": {
      "status": "UP",
      "details": {
        "cluster_name": "flowable-cluster",
[…]
```


GET /actuator/info


```
{
 "env": {
  "name": "DEV WORK 3.3.0-SNAPSHOT"
 },
 "git": {
  "commit": {
   "id": "0cf966d"
  }
 },
 "build": {
  "version": "3.3.0-SNAPSHOT",
  "artifact": "flowable-work-embedded-frontend-app",
  "name": "flowable-work-embedded-frontend-app",
  "number": "FLOW-FP-JOB1-3506",
  "group": "com.flowable.work",
  "time": "N/A"
 },
 "flowablePlatform": {
  "version": "3.3.0-SNAPSHOT"
 },
 "flowable": {
  "dbVersion": "6.5.0.1",
  "version": "3.3.0-SNAPSHOT"
 }
}
```

Logging

- Flowable uses the SLF4J adapter for logging
- Use the **SLF4J** Logger interface to log to the same stream as Flowable does
- Pipe your log files to a tool (i.e. Logstash) to alert on **WARN or ERROR events**

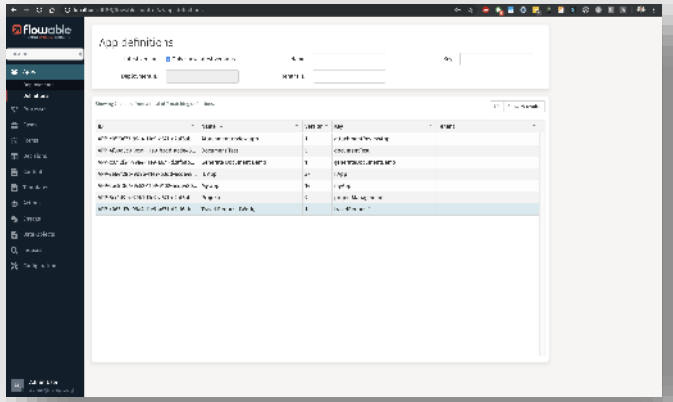
Pro-Active Alerting

- You can add listeners (see ’Hooks and Listeners’) to your application to call other system pro actively
- For example if a job fails
- But beware, this is fragile (what if the listener fails?)

Flowable Control

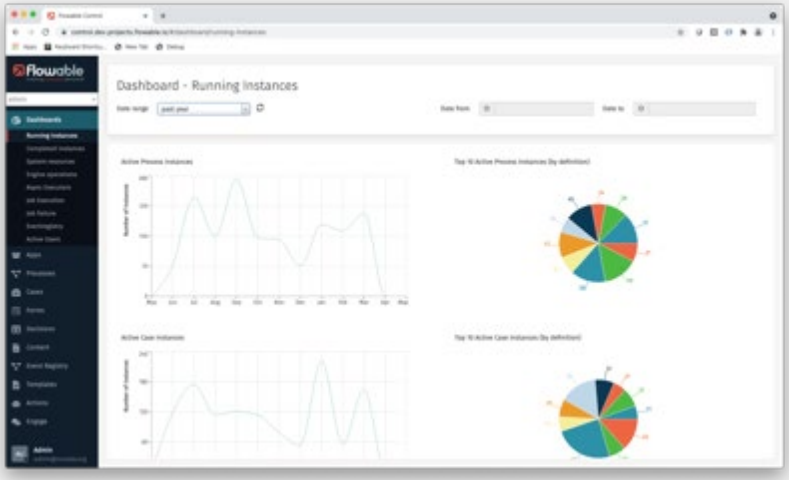
Overview

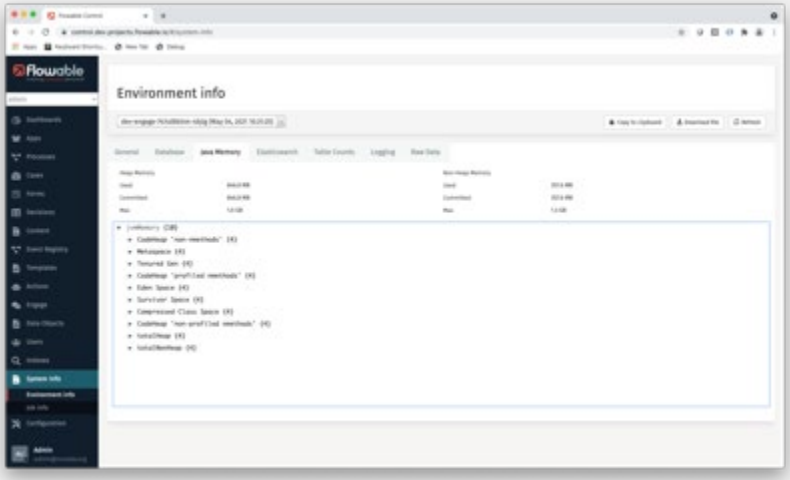
- Flowable Control is the **admin tool** for your Flowable applications
- **Main Features**:
  - System dashboard and support information
  - Overview of **apps**, **definitions** and **instances** (processes, cases, forms, templates, content etc.)
  - Update of **Variables**
  - Overview of **Elastic Search indices** and possibility to **reindex** them
  - Manage **deployments**
  - **Graphical migration** of single process instances
  - Overview and re-execution of **jobs**



Dashboard and System Information

- Insights into **health and load** of application
- Gathered information about **configuration**
- Possibility to configure async executors



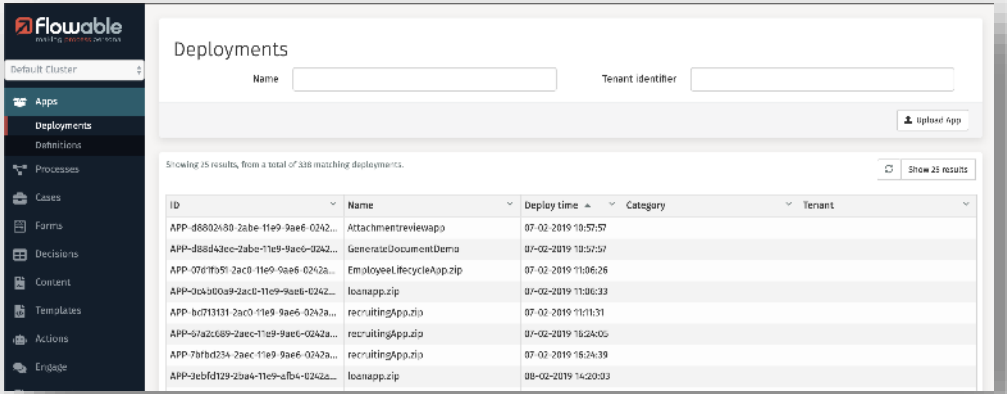


Deployments

- Flowable Control shows both, **App Deployments** and **Model Deployments** (e.g. of individual files as auto deployment)

Features:

- Possibility to make deployments
- Inspect deployment
- Browse through deployed definitions
- Delete deployments

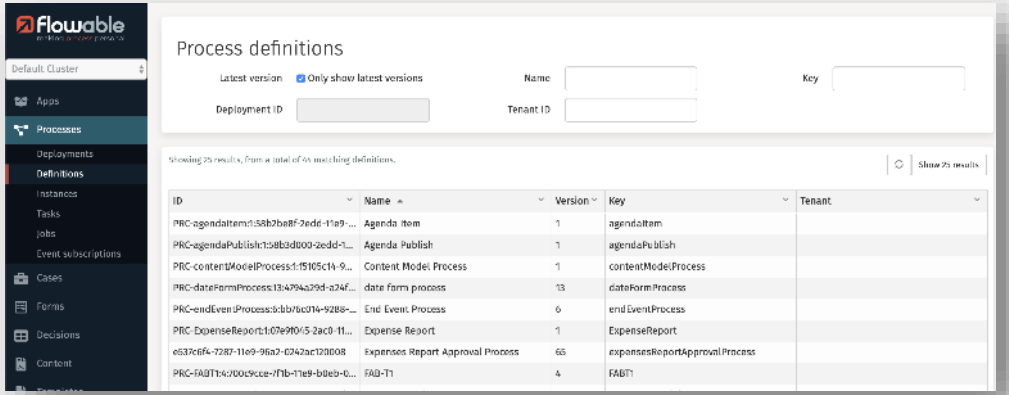


Definitions

In Flowable Control, you can inspect **Definitions** of Processes, Cases, Forms, Decisions, Content, Templates etc.

Features:

- Either display all or only the newest definitions
- Open definitions to see details and a
- Navigate to jobs, forms and other related entities
- Directly access instances of a given definition

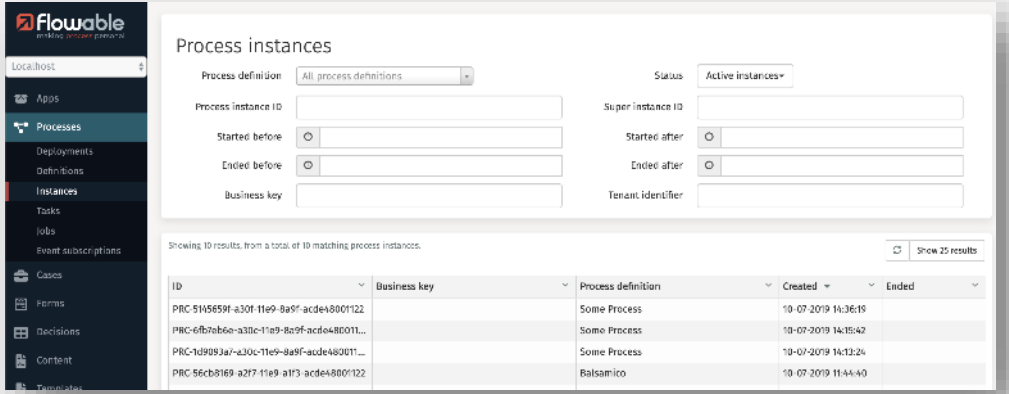


Instances

It is possible to search for and inspect individual **instances**.

Features:

- Search for instances by different criteria
- See the details (e.g. variables, tasks, jobs etc.) of each instances)
- Visualize the current state
- Perform actions, e.g.:
  - Terminate (=end) instance
  - Delete instance
  - Perform migration
  - Change state



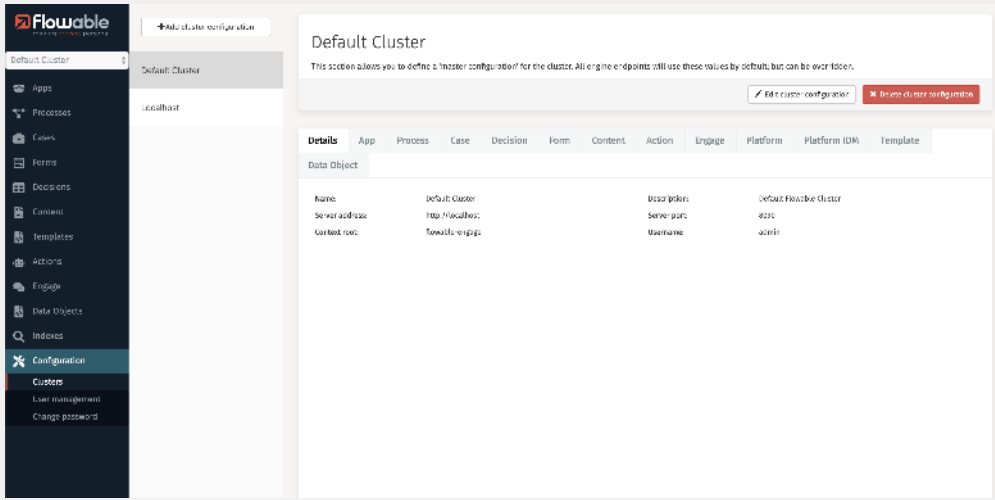
Jobs

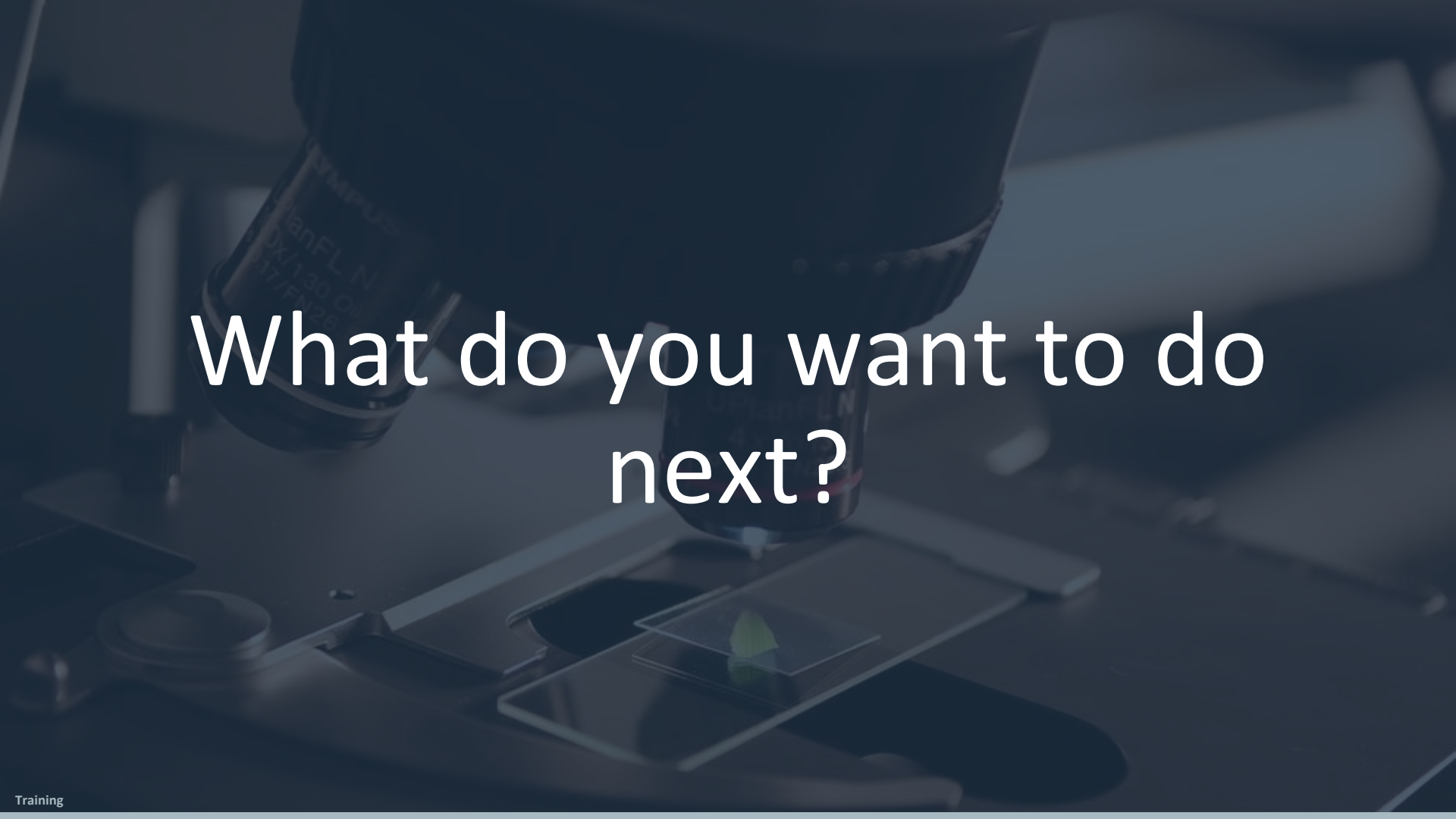
- Flowable employs the principle of **Job Queues** in different areas:
  - Each service task call results in an **executable job**
  - Timer events and async service tasks create **timer jobs**
  - Jobs can be suspended or, are moved to the **deadletter queue** if they fail to often

- In Flowable Control you can:
  - **Retry** jobs (e.g. after an issue has been fixed)
  - **Move/clear** jobs
  - **Inspect** jobs to find exceptions or other issues

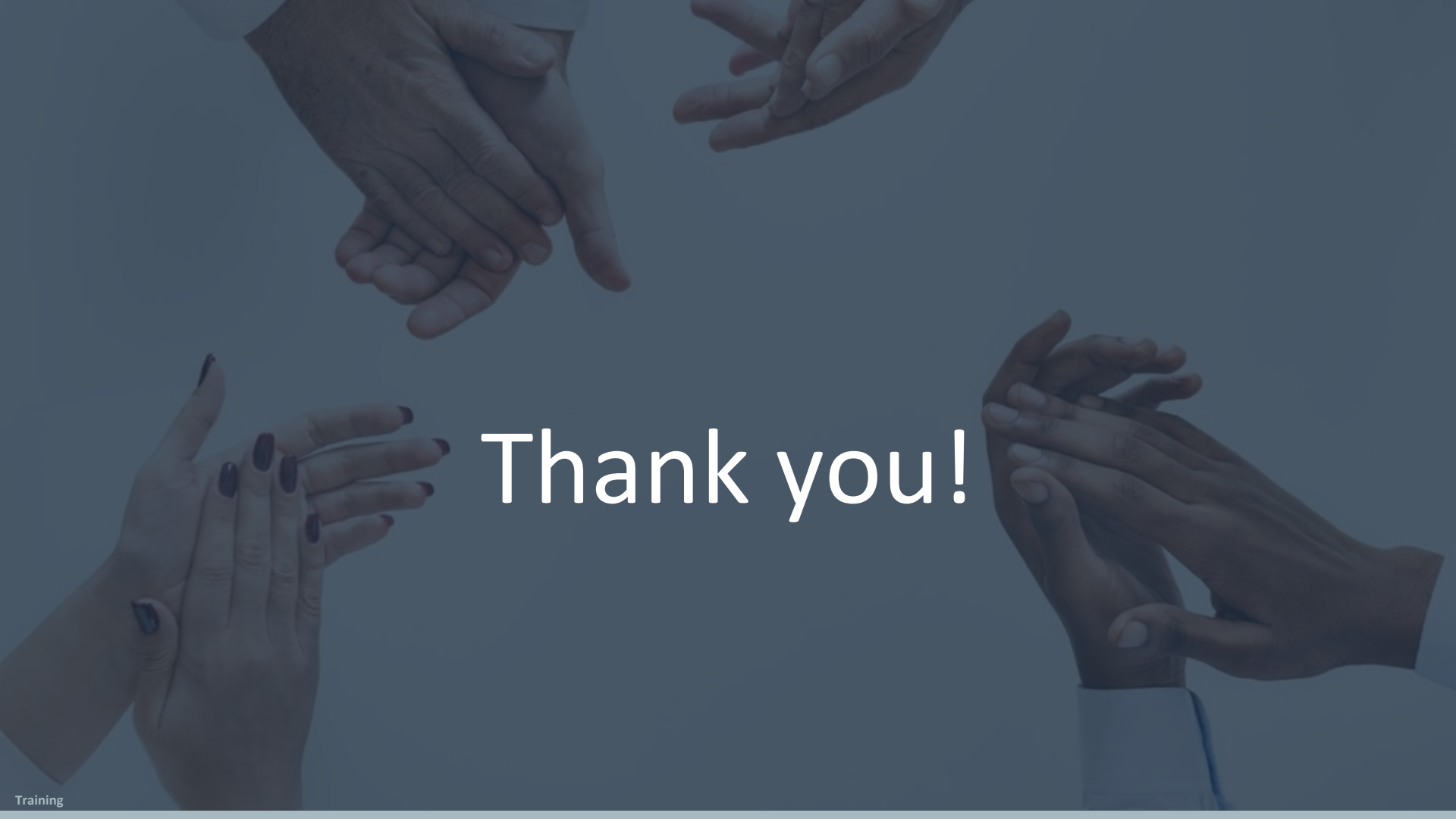
Configuration

- Set up **multiple clusters** (=environments)
- Possibility to individually configer each API separately
- Configuration:
  - Name
  - Server Address
  - Context Root
  - Port
  - User name and password





What do you want to do next?


Thank you!


Flowable
making process personal