



Flowable Trainings



Developing for Flowable Platform

A photograph of two business professionals in an office setting. A man in a blue and white checkered shirt is shaking hands with a woman whose blonde hair is visible. They are standing behind a desk. On the desk, there is a laptop displaying a presentation slide titled "Company's Growth" with a world map graphic, and another laptop partially visible. A small potted plant sits on the right side of the desk.

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**

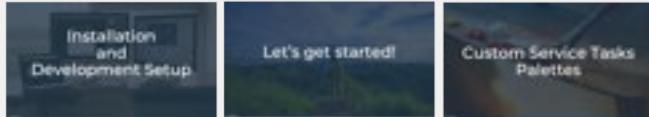
- Set up your first Flowable Work application locally
 - Create a simple workflow with a service task
 - Make the service task customizable

- **Learn how to work with the Flowable APIs**

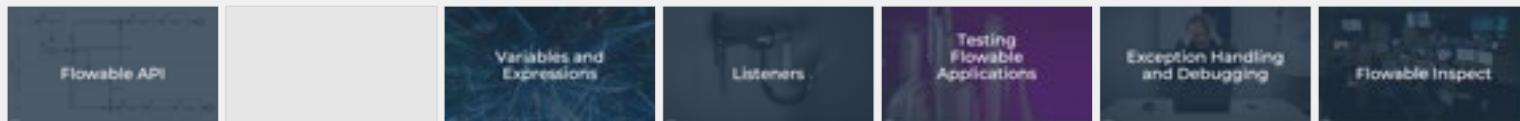
- Start, query and inspect your Engines using Runtime, Repository, History and Management Services
 - Learn how to extend and integrate your Flowable Applications

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>

Flowable Training

Repository can be found here:

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

Project:

Unzip this project anywhere on your dev machine.



flowable-training...-started-main.zip

License:

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



flowable.license

Settings XML:

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



settings.xml

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.hanson.design.TrainingHansonDesignApplication`
`com.training.hanson.work.TrainingHansonWorkApplication`
`com.training.hanson.control.TrainingHansonControlApplication`
- 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

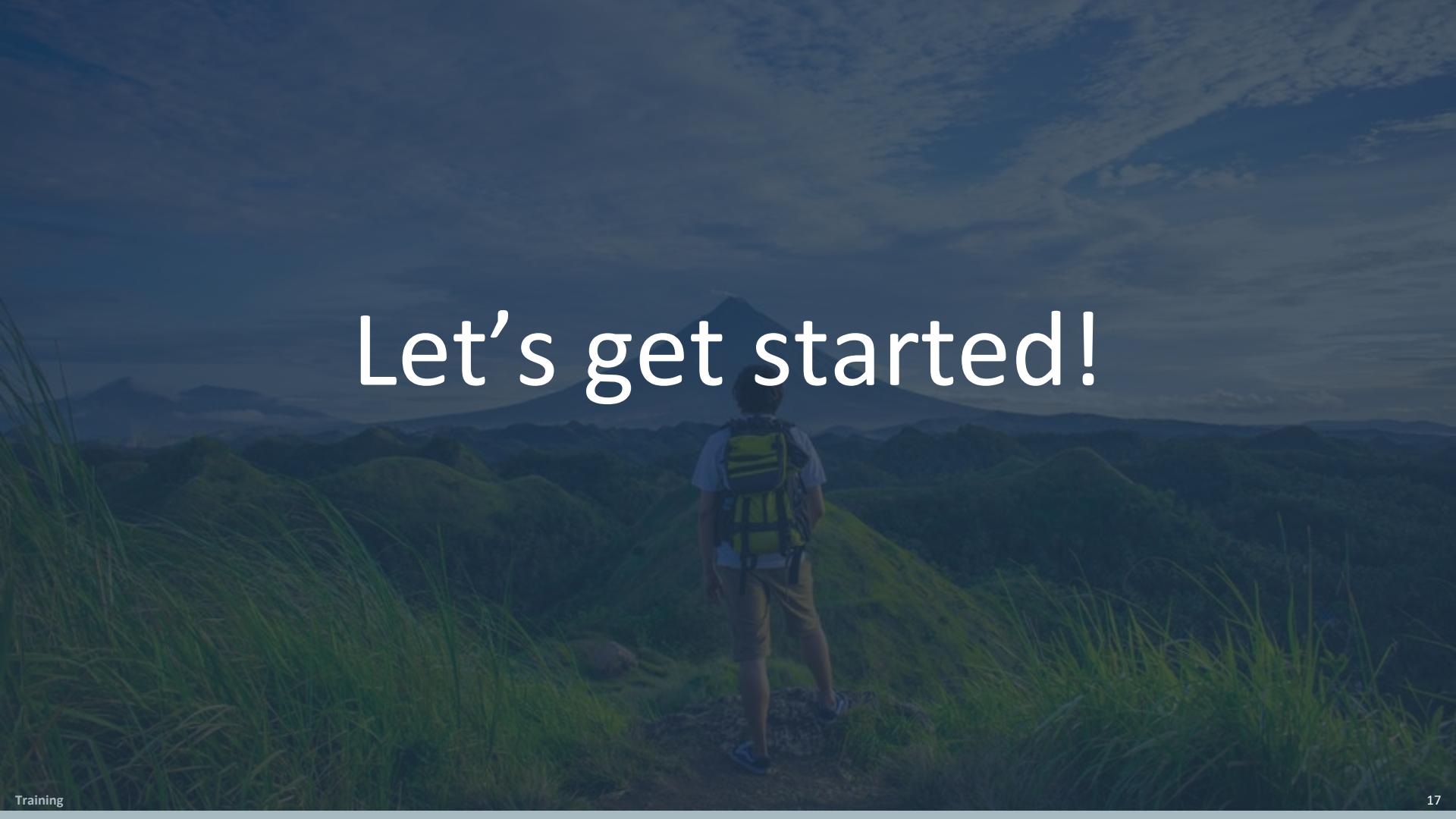
- Move your license file to `~/.flowable/flowable.license`

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

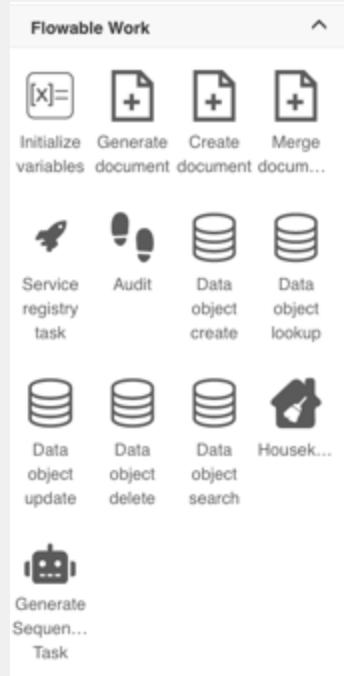
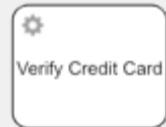
A photograph of a person from behind, wearing a backpack, standing on a grassy hillside. They are looking towards a range of mountains in the distance under a cloudy sky.

Let's get started!

A photograph of a person in a coffee shop. They are holding a white mobile payment terminal with a blue screen and a card slot. Their hands are positioned over the device. In the background, there's a counter with various items: a small sign that says "KONA COFFEE CO.", a cup of coffee with a latte art swirl, a jar labeled "Fall Spice", a small bottle of spice, and a stack of books. A person in a dark t-shirt is standing behind the counter. The scene is lit with warm, natural light.

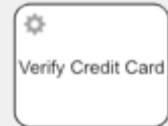
Create a Service Task

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

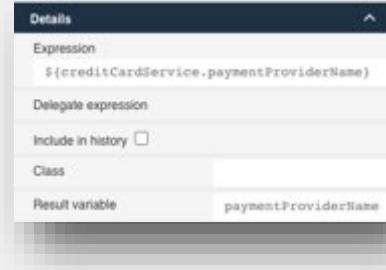


- Service tasks represent **work done by a computer**.
- Usually, it calls **Java code** and can involve external systems.
- There are four ways of declaring the Java logic to invoke:
 - Evaluating a **value expression**
 - Invoking a **method expression**
 - Class extending an **AbstractPlatformTask**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

Service Task JUEL Expressions

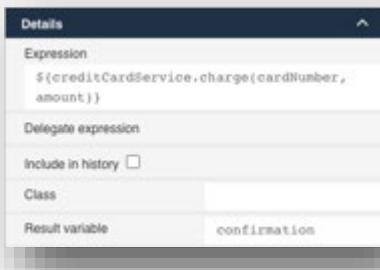
- Evaluating a value expression

```
<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

The diagram illustrates the mapping between a BPMN task and its corresponding Java code. On the left, a BPMN process is shown with a task named "Verify Credit Card". A red arrow points from this task to a code editor window below. The code editor displays Java code for a class named `CreditCardValidationTask`, which extends `AbstractPlatformTask`. The code includes annotations `@Component("creditCardValidation")` and `@Override`, and a method `executeTask` that takes a `VariableContainer` parameter. Another red arrow points from the `VariableContainer` parameter to the "Variables" tab of the process instance details at the bottom. On the right, three boxes represent Java classes: `JavaDelegate`, `PlanItemJavaDelegate`, and `AbstractPlatformTask`. `JavaDelegate` and `PlanItemJavaDelegate` both have dashed arrows pointing to `AbstractPlatformTask`, indicating it implements these interfaces.

Details

Expression

Delegate expression
\${creditCardValidation}

Include in history

Class

Result variable

Verify Credit Card

`@Component("creditCardValidation")`

`public class CreditCardValidationTask extends AbstractPlatformTask {`

`@Override`

`public void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer) {`

`}`

`}`

Details Variables Tasks Subprocesses Jobs Event subscriptions Decisions Form instances History

This process instance has 4 variables.

Name	Type	Value
cardNumber	String	1234 5678 9012 3456

JavaDelegate

PlanItemJavaDelegate

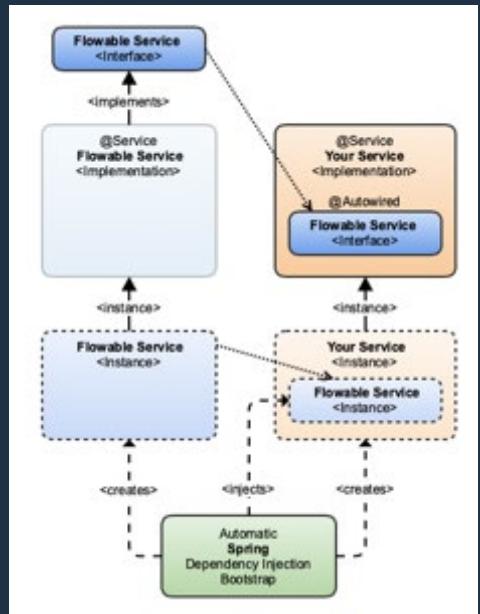
implements

AbstractPlatformTask

```
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

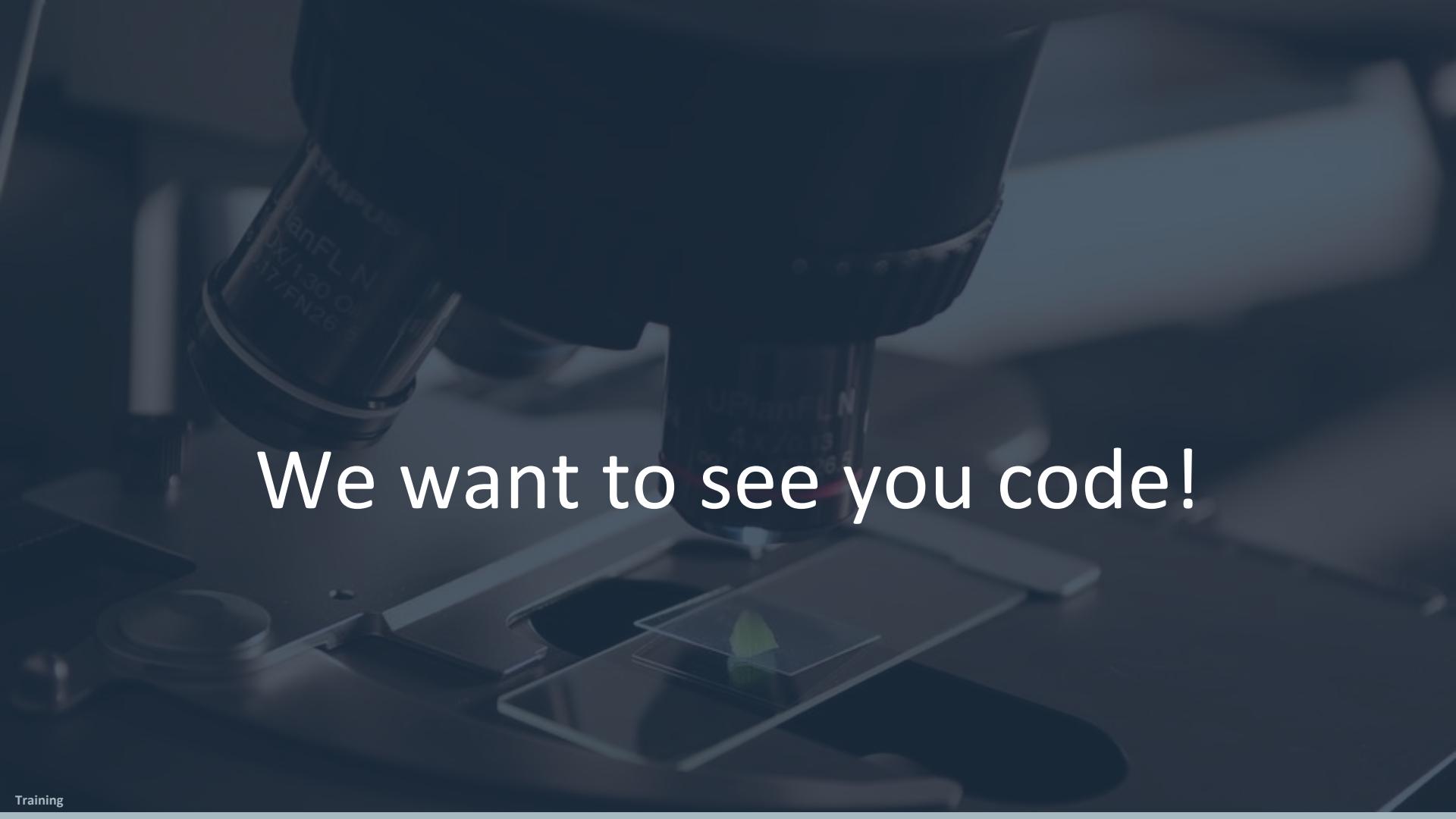
Let's do it the Flowable way.



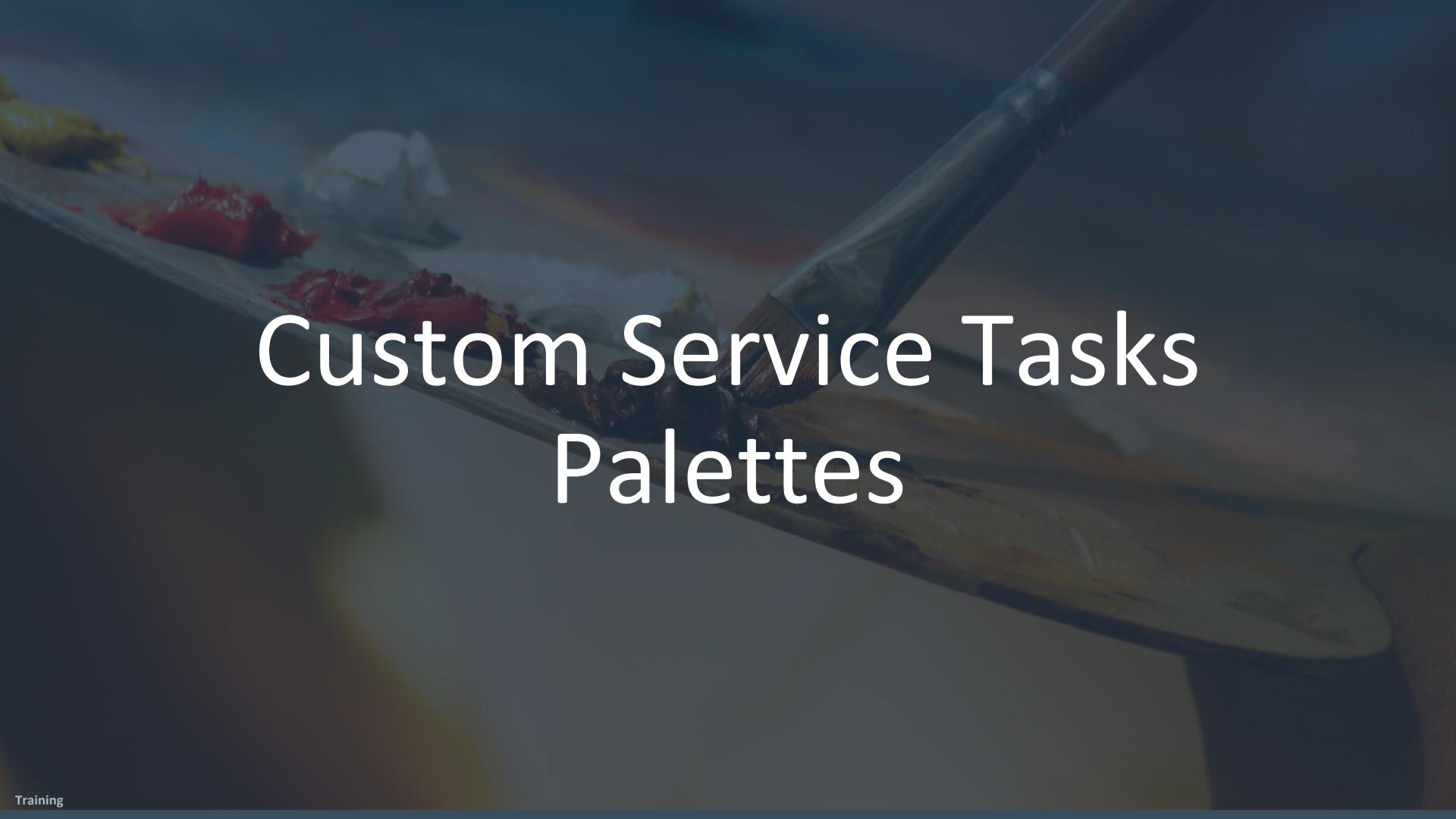
Goal:

- 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!

A close-up photograph of a compound light microscope. The eyepiece lens on the left is labeled 'Olympus' and 'PlanFL N 10x 130 OI.2 17/FN26'. The objective lens on the right is labeled 'UPlanFL N 40x 130 OI.1 17/FN26'. A glass slide with a small, green, irregularly shaped sample is positioned on the stage. The background is dark.

We want to see you code!

A close-up photograph of a painter's palette. The palette is light-colored wood and holds several dollops of paint in various colors: yellow, red, blue, and green. A paintbrush with a dark handle and silver ferrule lies diagonally across the palette. The background is dark and out of focus.

Custom Service Tasks Palettes

Recap and Motivation

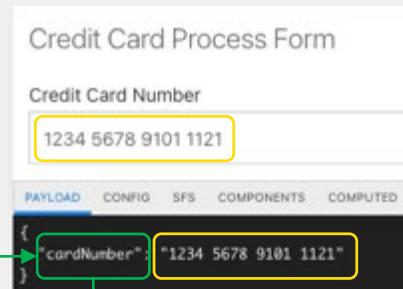
- Service Tasks are there to execute **Java Code** in BPMN and CMMN
- Challenges
 - How to make them usable for **Citizen Developers**?
 - How to keep the **Process Knowledge** outside of the code?
 - How to make them reusable **across** different models?

Tightly coupled Model and Code

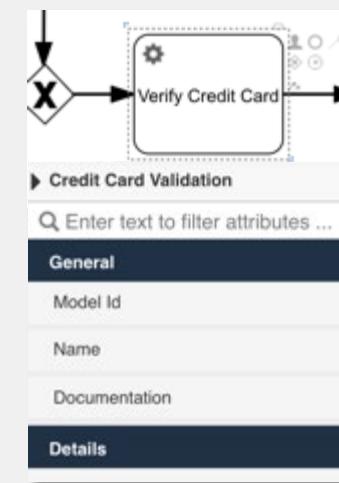
Model



Instance

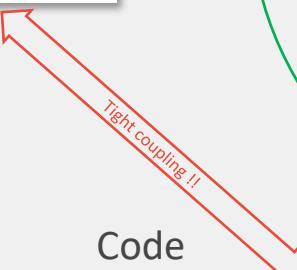


Service Task



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

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**.

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  
        },  
        {  
          "id": "card-expiration-date",  
          "category": "edoras",  
          "type": "SimpleTextExpression",  
          "index": 11,  
          "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/>

The image shows two side-by-side screenshots. On the left is a JSON schema configuration for a palette item. On the right is a stencil editor interface.

JSON Schema (Left):

```
{
  "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
        }
      ]
    }
  ]
}
```

Stencil Editor (Right):

The stencil editor displays a task named "Credit Card Validation". It includes fields for "Model Id" (creditCardValidationTask), "Name" (Credit Card Validation), and "Documentation". Under the "Details" tab, there is a "Credit Card Number" field with the value "\${cardNumber}". The "Execution" tab shows options for "Asynchronous" (unchecked), "Execution listeners", "Skip expression", and "Is for compensation" (unchecked).

Creating a Palette

The diagram illustrates the process of creating a palette for a service task. It consists of three main components:

- Service Task Configuration:** On the left, a screenshot of the Flows application interface shows the configuration for a "Verify Credit Card" service task. The "Details" tab is selected, displaying properties like "Delegate expression" set to "\${creditCardValidation}" and "Execution" settings.
- JSON Configuration:** In the center, a code editor displays JSON configuration for a "CreditCardValidationTask". Three specific sections are highlighted with red boxes:
 - The "id": "CreditCardValidationTask" section, which defines the task's ID and super ID.
 - The "properties" section, which contains configurations for "credit-card-number", "card-valid-result-variable-name", and "delegateexpression".
 - The "visible": false setting under "delegateexpression".
- Palette Item Preview:** On the right, a preview window titled "Credit Card Validation" shows the resulting palette item. It includes the task's name, a description ("Credit Card Validation"), and its configuration details: "Credit Card Number" (value: \${cardNumber}), "Card Valid Result Variable Name" (isValid), and "Execution" settings.

Red arrows indicate the flow from the service task configuration to the JSON configuration, and from the JSON configuration to the palette item preview, illustrating how the configuration is mapped to the final palette item.

```
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
```

group.scooter.title=Scooter Tasks
CreditCardValidationTask.title=Credit Card Validation
property.credit-card-number.title=Credit Card Number
property.card-valid-result-variable-name.title=Card Valid Result Variable Name

Credit Card Validation

Enter text to filter attributes ...

General

Model Id creditCardValidationTask1

Name Credit Card Validation

Documentation

Details

Expression

Delegate expression \${creditCardValidation}

Include in history

Class

Result variable

Store as local variable

Class fields

Triggerable

Exception Mappings

Execution

Asynchronous

Execution listeners

Skin expression

Training

group.scooter.title=Scooter Tasks
CreditCardValidationTask.title=Credit Card Validation
property.credit-card-number.title=Credit Card Number
property.card-valid-result-variable-name.title=Card Valid Result Variable Name

Credit Card Validation

Enter text to filter attributes ...

General

Model Id creditCardValidationTask1

Name Credit Card Validation

Documentation

Details

Credit Card Number
\${cardNumber}

Card Valid Result Variable Name
isValid

Exception Mappings

Execution

Asynchronous

Execution listeners

Skip expression

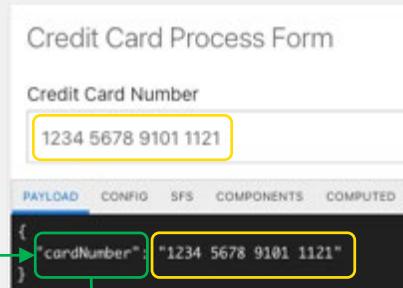
Is for compensation

Tightly coupled Model and Code

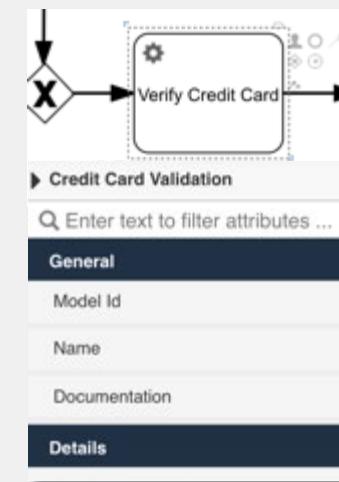
Model



Instance



Service Task



Code

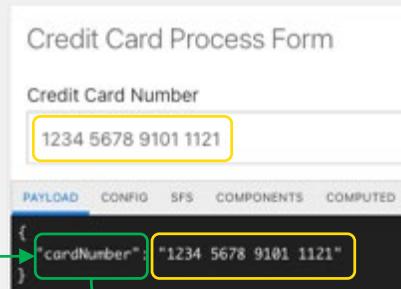
```
@Override  
public void executeTask(VariableContainer variableContainer, ExtensionElementsContainer extensionElementsContainer) {  
    String cardNumberValue = (String)variableContainer.getVariable("cardNumber");
```

Loosely coupled Model and Code

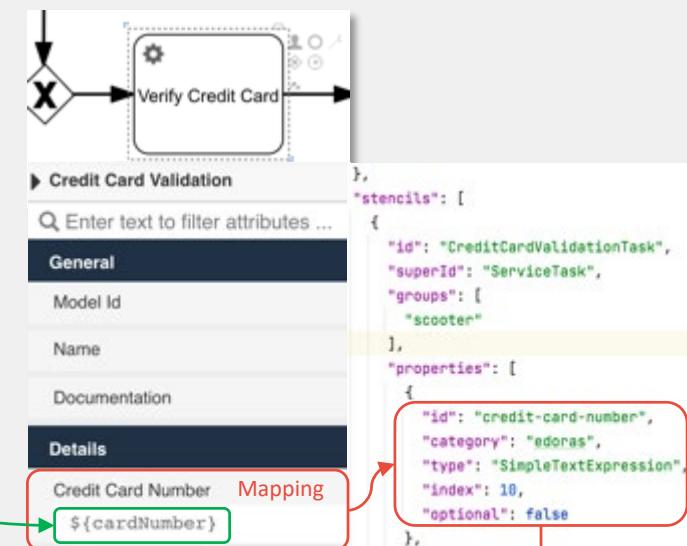
Model



Instance



Service Task with ExtensionElement

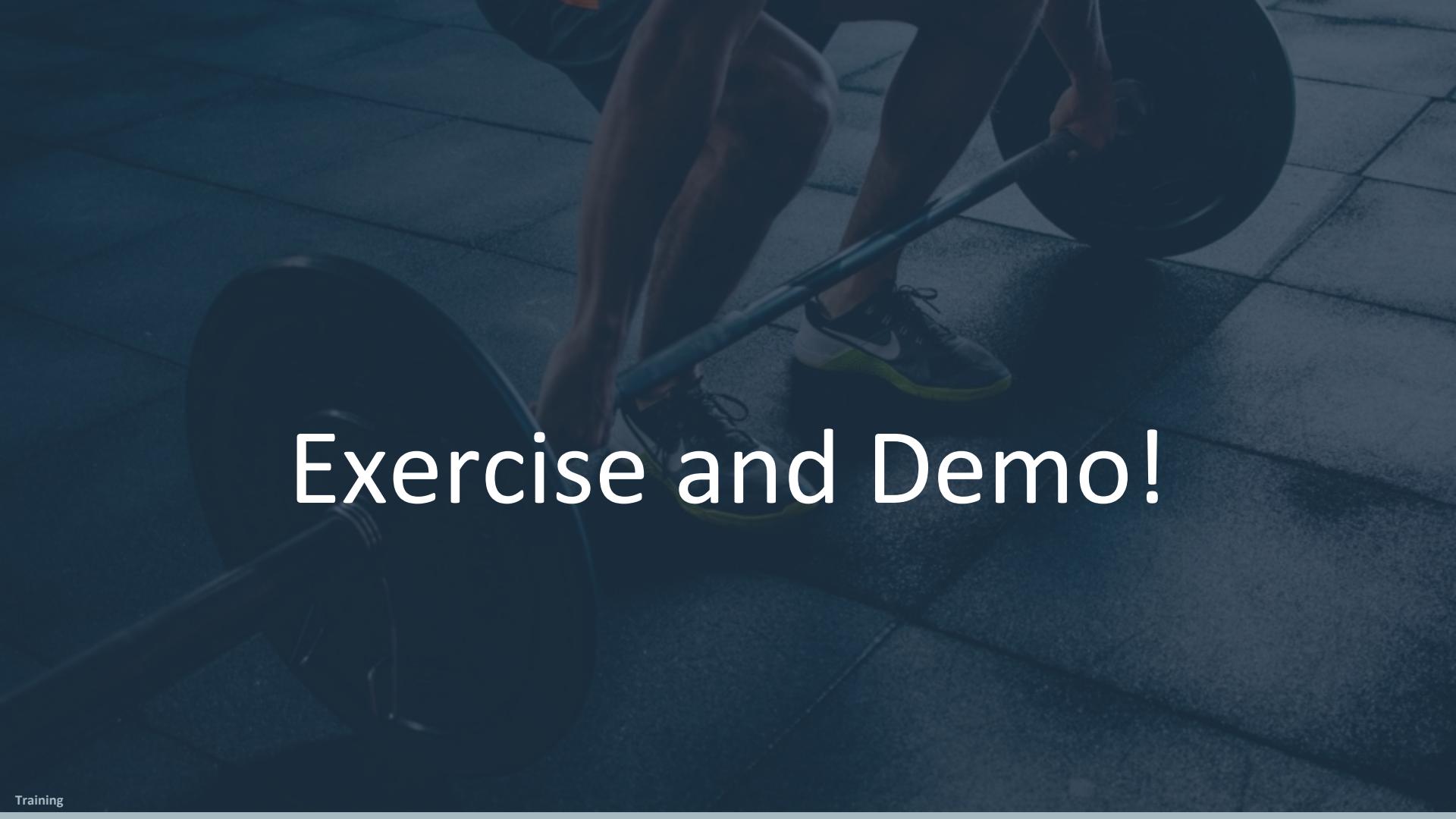


Code

```
@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

A dark, slightly blurred background image of a person performing a deadlift with a barbell. The person is wearing dark athletic clothing and grey sneakers with yellow accents. The barbell has large black weight plates. The scene is set on a light-colored wooden floor with some markings.

Exercise and Demo!

Hello World

Let's do it the Flowable way.



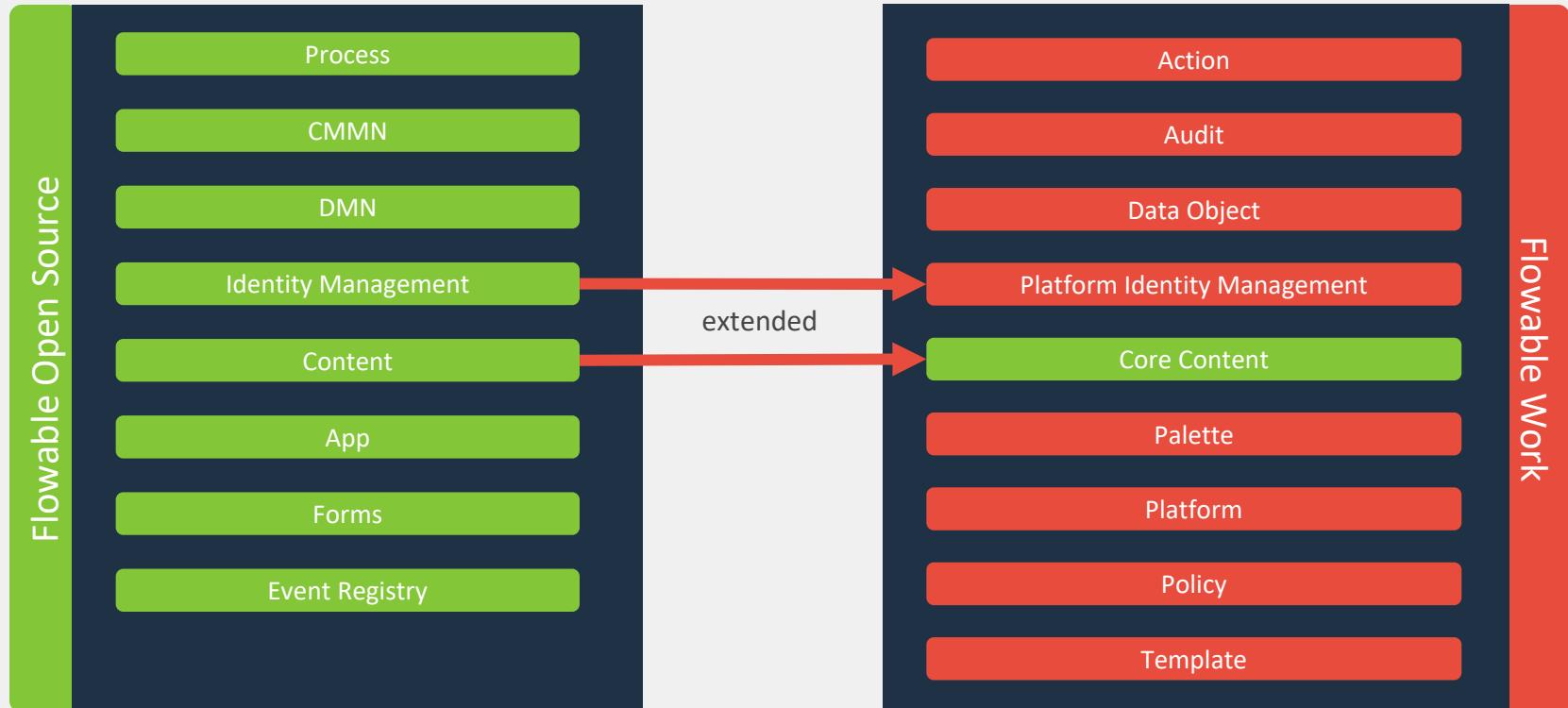
Goal:

- 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

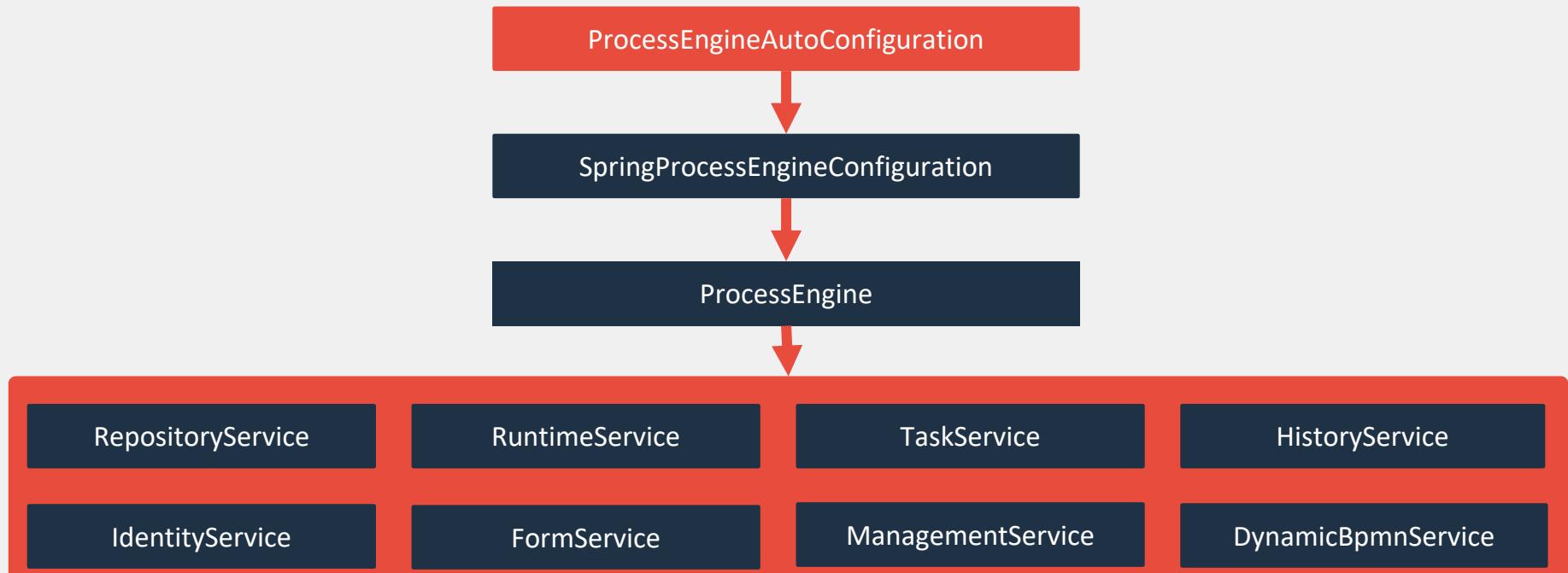
List of Engines



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

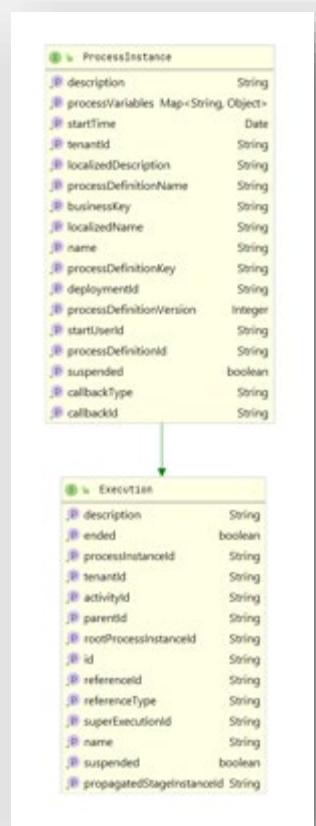
- ACT_CMMN_RU_*:
Runtime. Data of living case instances, plan items and sentries.
- ACT_CMMN_HI_*:
History. Historic data of case instances and plan items.
- ACT_CMMN_CASEDEF:
Case definitions
- ACT_CMMN_DEPLOYMENT:
CMMN Deployments and resources

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**



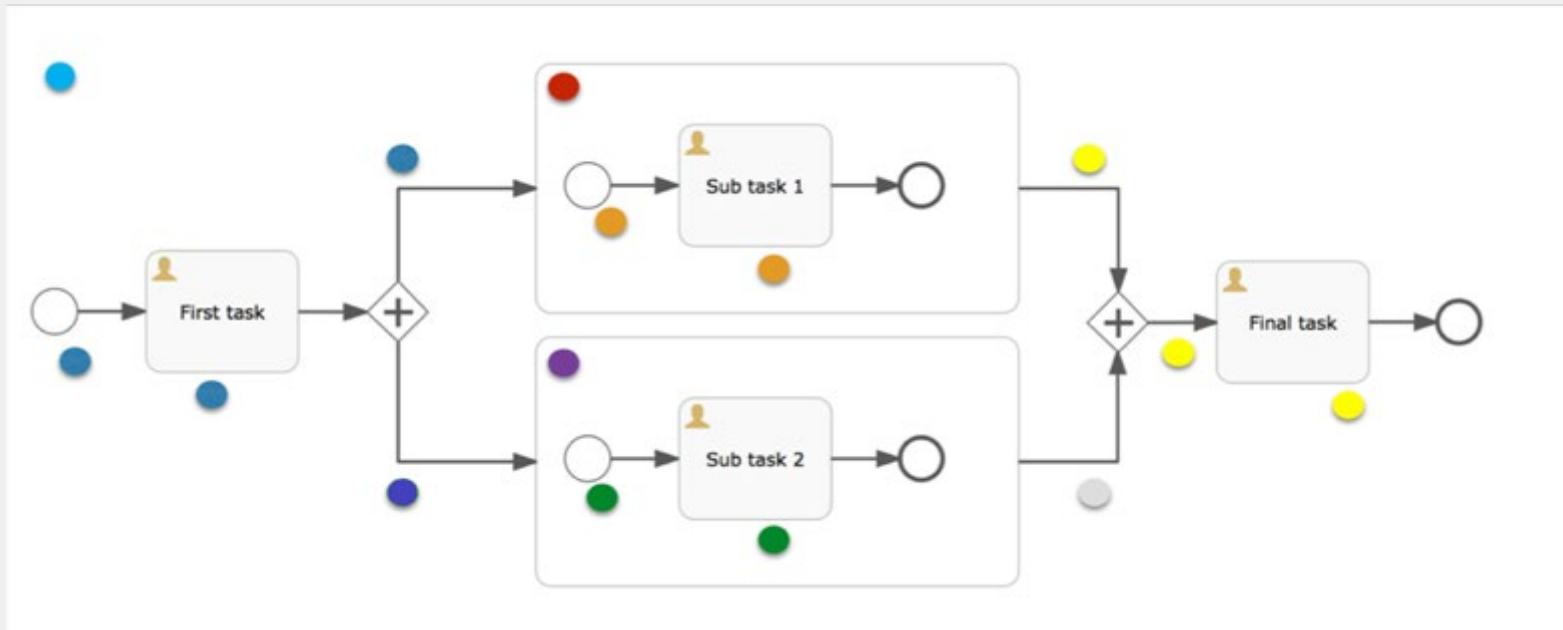
Executions

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}`

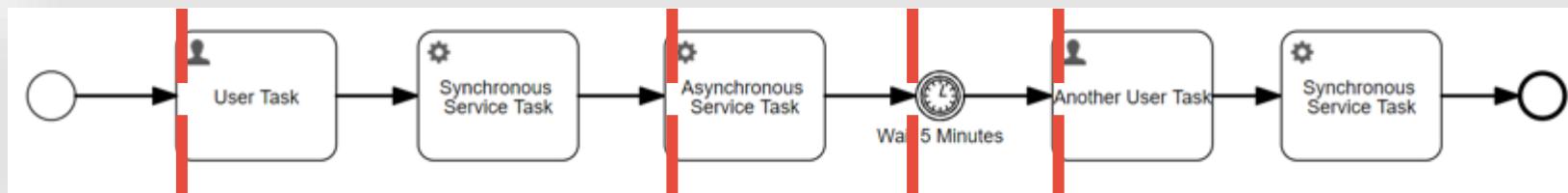
Execution tree

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**".

```
int ilength, i;
double dblTemp;
bool again = true;

while (again) {
    iN = -1;
    again = false;
    getline(cin, sInput);
    system("cls");
    stringstream(sInput) >> dblTemp;
    ilength = sInput.length();
    if (ilength < 4) {
        again = true;
        continue;
    } else if ((sInput[ilength - 3] >= 'A') &
               (sInput[ilength - 3] <= 'Z')) {
        again = true;
        continue;
    } while (i < ilength) {
        if ((sInput[i] == (sInput[ilength - 3]))) {
            if ((iN == (ilength - 3))) {
                cout << "The first character of the string is a native field." << endl;
            }
        }
        i++;
    }
}
```

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

The diagram illustrates five variable types in a JSON object:

- Native Field:** "id": "PRC-e5834f60-3ff6-4bc1-8235"
- List Variable:** "damagedParts": ["Driver Seat", "Window"]
- JSON Variable:** "analysis": {"text": "Clearly something is broken.", "estimatedCost": 1200}
- Binary Variable:** "someBinaryVariable": (hexadecimal string)

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.

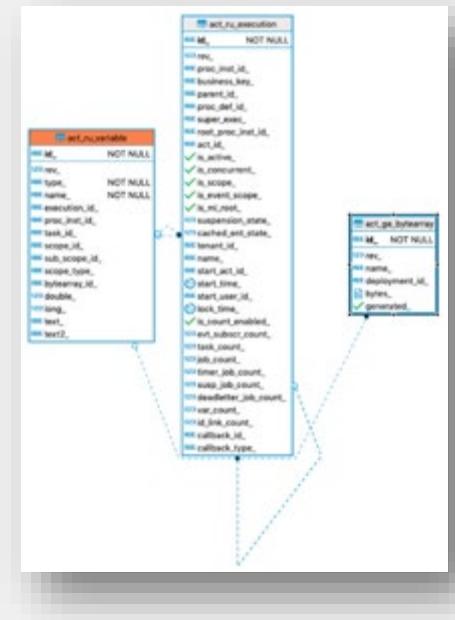
How to store variables

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)}
```

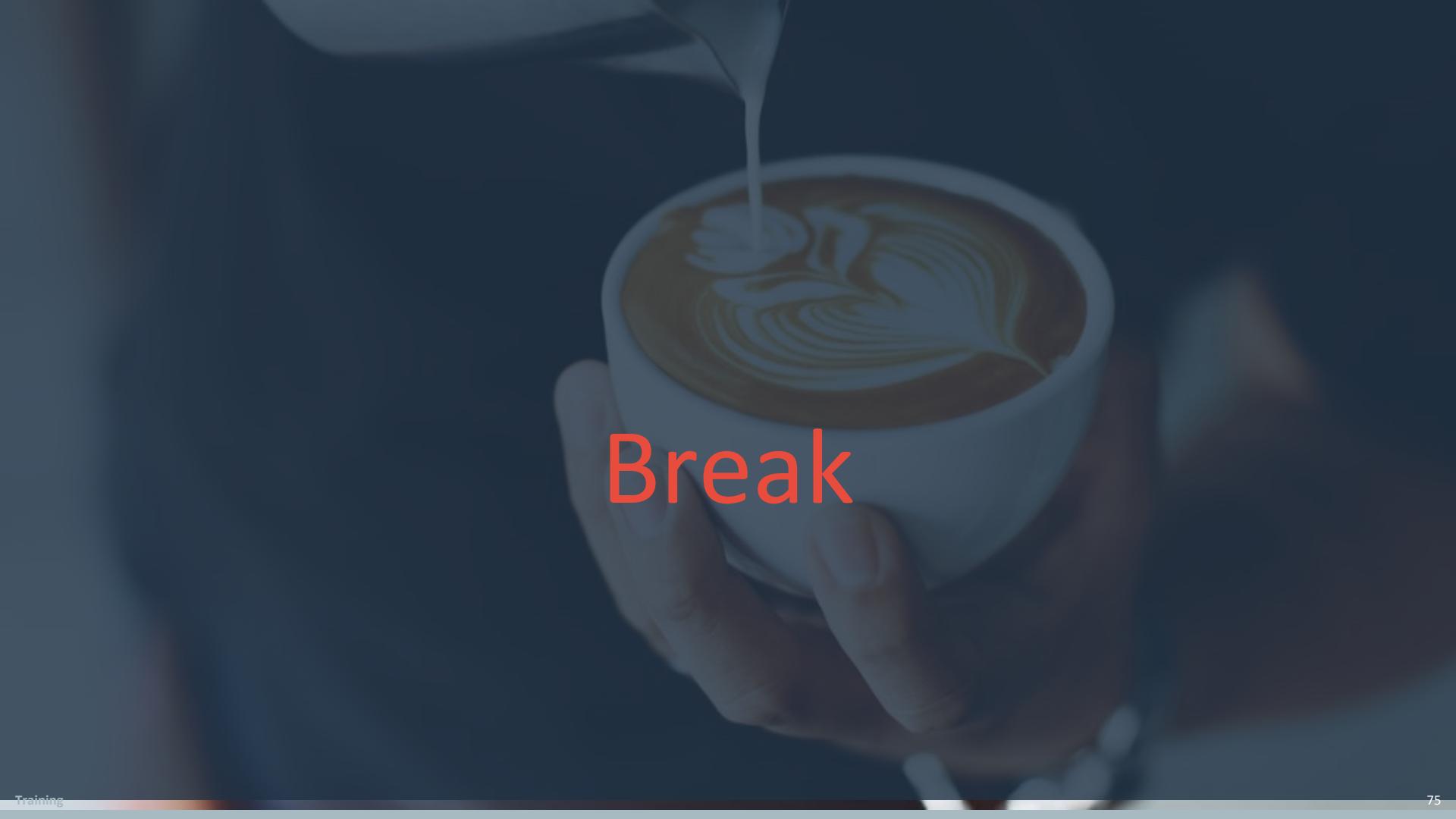
Many more...

- **Aliases:** **vars** or **var** can be used instead of **variables**
- The execution/case instance is automatically injected.

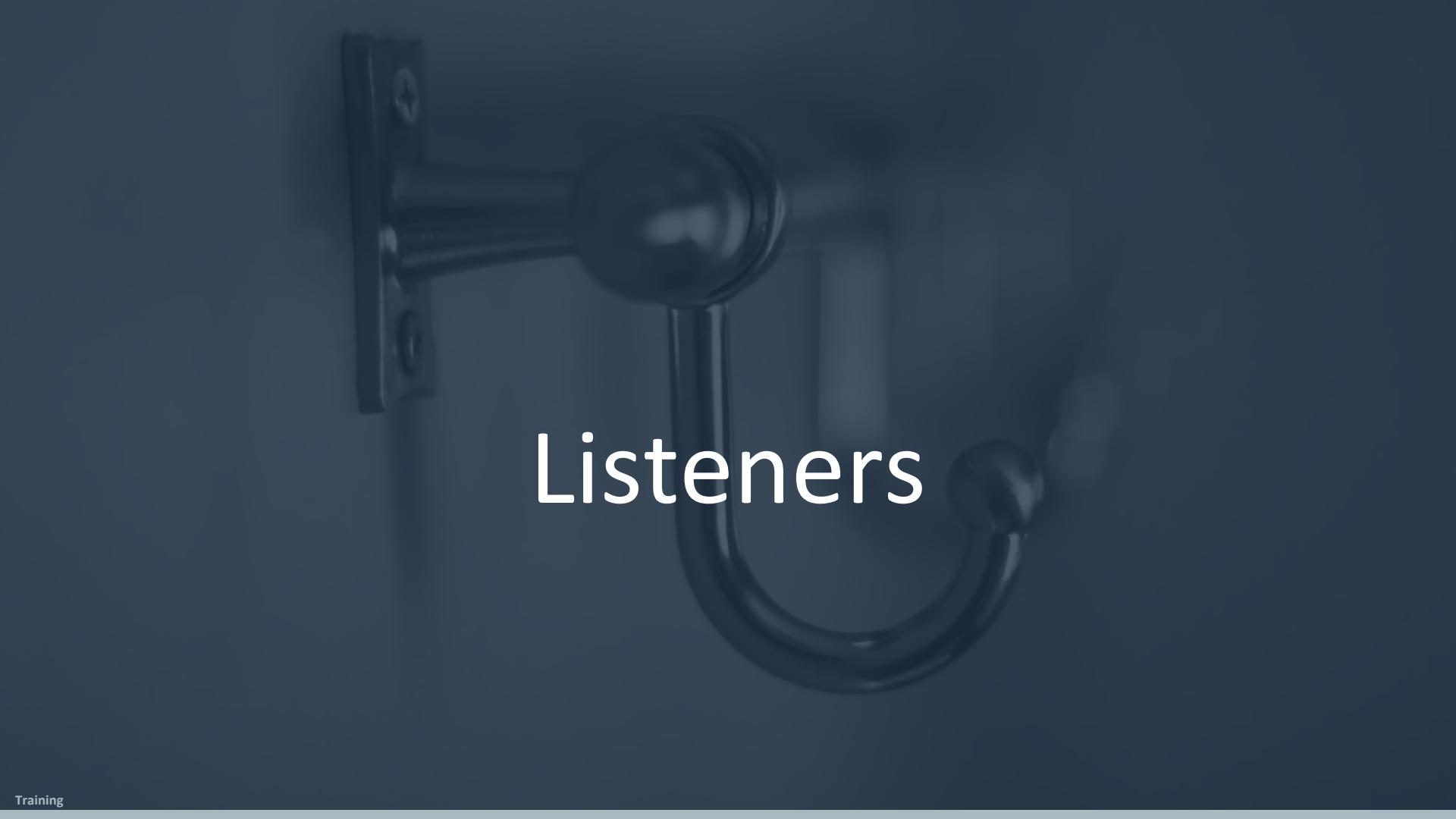
Flw Backend Expressions

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

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

A close-up photograph of a person's hand holding a white ceramic cup. A stream of white milk or cream is being poured from above into the cup, creating intricate latte art patterns on the surface of the coffee. The background is dark and out of focus.

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**)

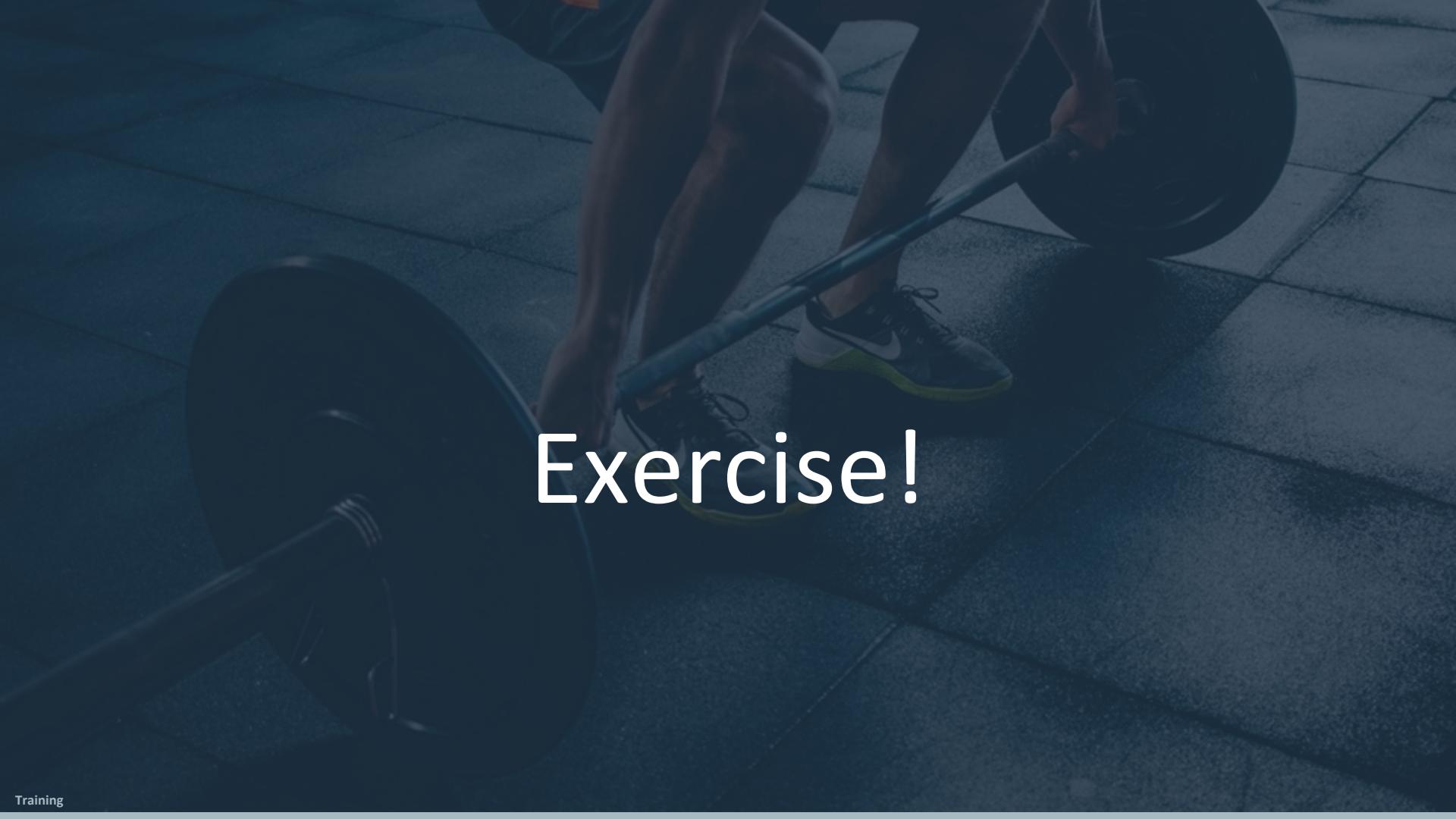
- Recommendation:**
Use expressions and call spring service

Execution listeners						
	* Event	Class	Expression	Delegate e...	Fields	
	End	com.flowable...				
	Start		\$(executionLoggerBean.logExecutionStart(execution))		Field name: fieldOne, String value: Any value, Express...	
	Start					

```
@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"));
    }
}
```

A person is performing a deadlift with a barbell. The person is wearing a dark t-shirt, dark shorts, and grey athletic shoes with yellow accents. The barbell has two large black weight plates on each side. The background is a gym floor with white chalk lines.

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

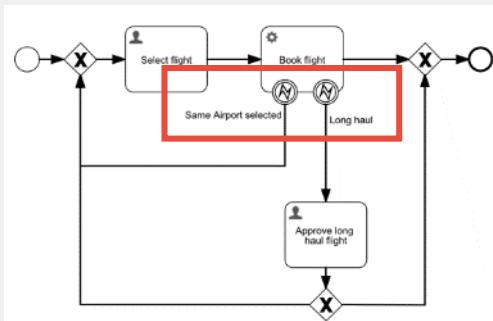
Any unexpected failures that should be mostly of technical nature. Must be handled via the the (REST) API or a tool like Flowable Control



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

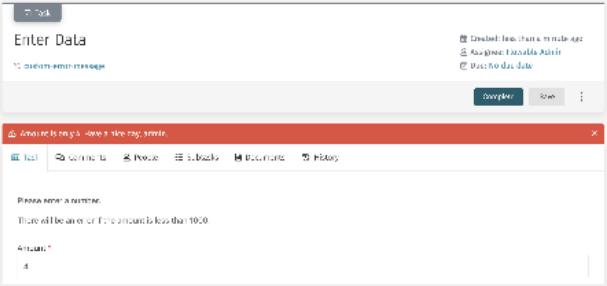


Details

Error code	longHaul
------------	----------

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 be configurable.
 - From the queue, move the job through Java or REST API or via Flowable Control.



Flowable Control

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

Edit cluster configuration

Name	Training Cluster
Description	Training Cluster
Server address	http://localhost
Server port	8091
Context root	flowable-work
Username	admin
Password
URL preview	http://localhost:8091/flowable-work/

Cancel Save cluster configuration

Update variable 'flightServiceAvailable'

Value	<input checked="" type="checkbox"/> true
-------	--

Cancel Update

Training Cluster

App Engine Process Engine Deployments Definitions Instances Tasks Jobs Event subscriptions CMN Engine

jobs

Job type: Deadletter jobs

Process instance ID:

Due before:

Exception: Only show jobs with an exception

Process definition: All process definitions

Tenant identifier:

Due after:

Show 1 results, from a total of 1 matching jobs.

ID	Due date	Process definition	Retries	Exception
JOB-09e725b-8799-11e9-b1ff-bab5181a0783	2019-06-05T15:52:2...	book-company-flight	0	Error while evaluating expression: \${flightSer...

Show 25 results

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!

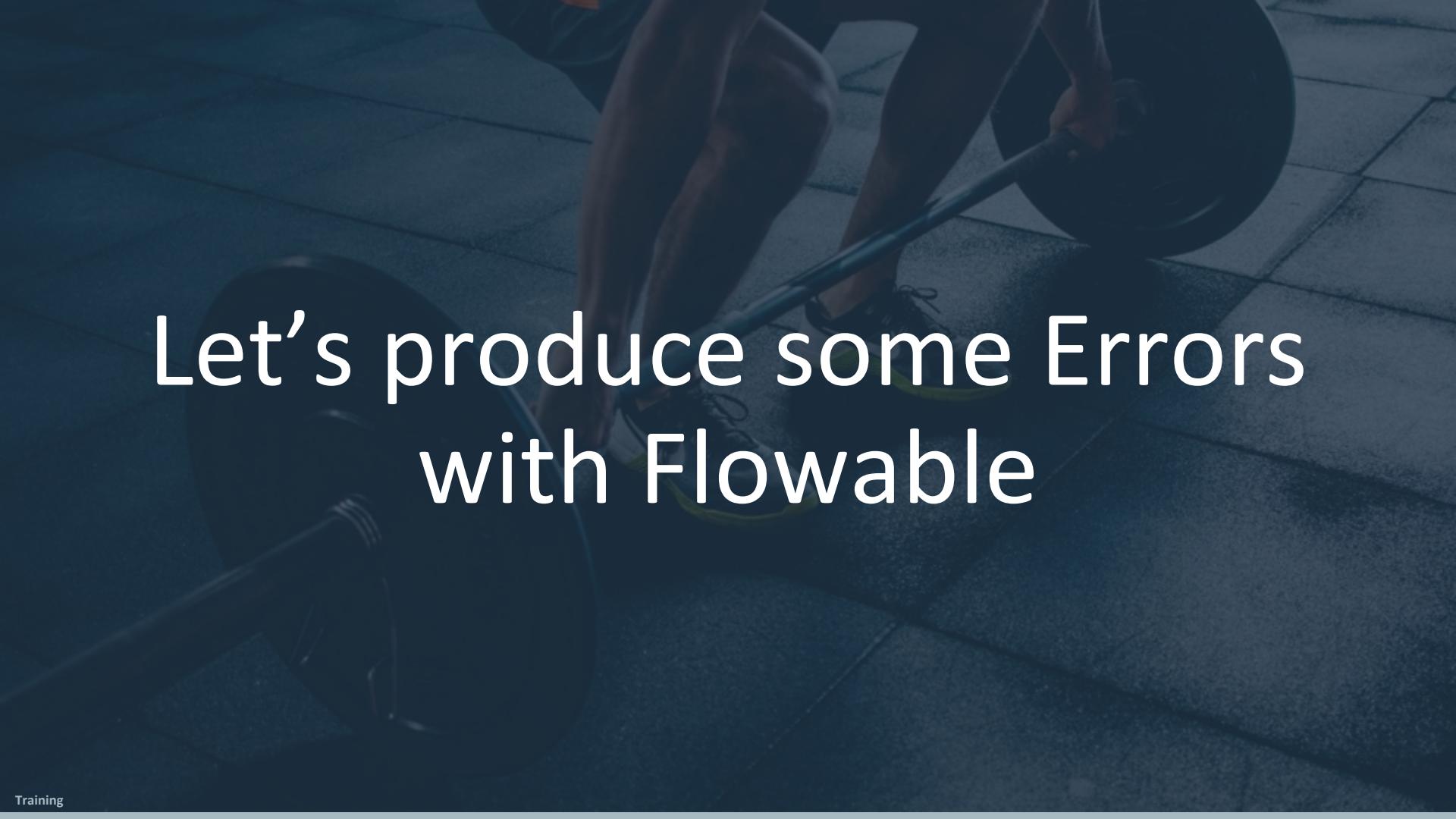
- When an activity (e.g. ServiceTask) is flagged as **asynchronous**, an artificial **wait state** is introduced.
- The job executor performing the previous step **commits** the transaction and creates an **async job** in the database for the next step.
- The **job executor** polls the database for new 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

A person is performing a deadlift with a barbell. The person is wearing a dark t-shirt and light-colored shorts. The barbell has weight plates on both ends. The background is a gym floor with white chalk marks.

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
 - Awaityility 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")
```

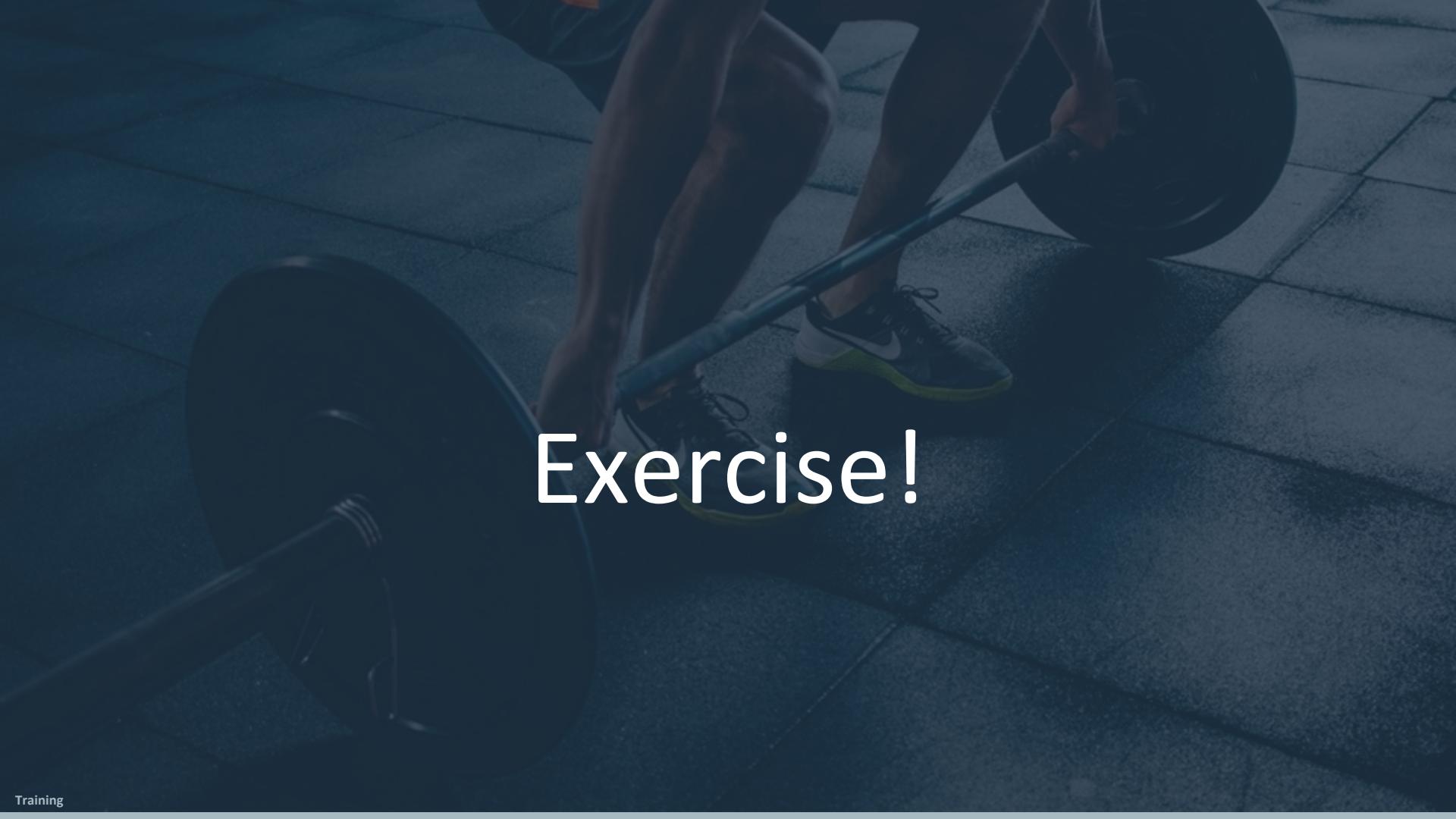
The test is now running as Spring Boot App and uses the default test configuration.
You can now **inject engines and services** as normal, e.g. through **@Autowired**.

- **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());
```

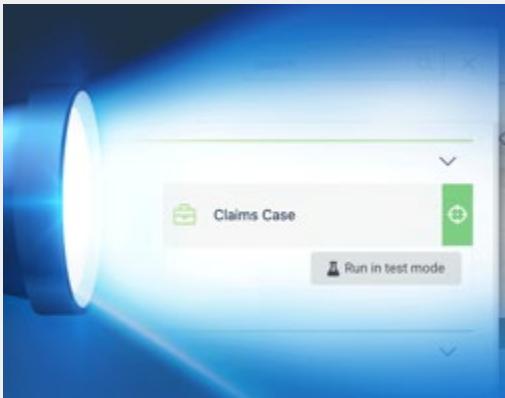
A person is performing a deadlift with a barbell. The person is wearing a dark t-shirt, dark shorts, and grey athletic shoes with yellow accents. The barbell has two large black weight plates on each side. The background is a gym floor with white chalk lines.

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

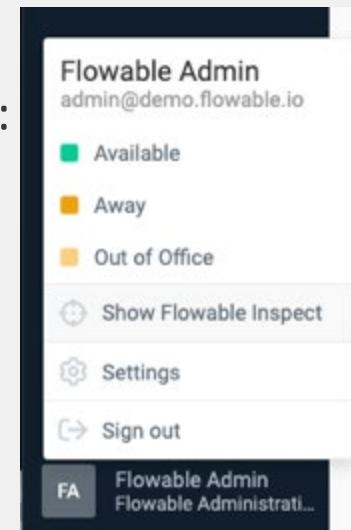
- Add Inspect Dependency to your classpath:

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

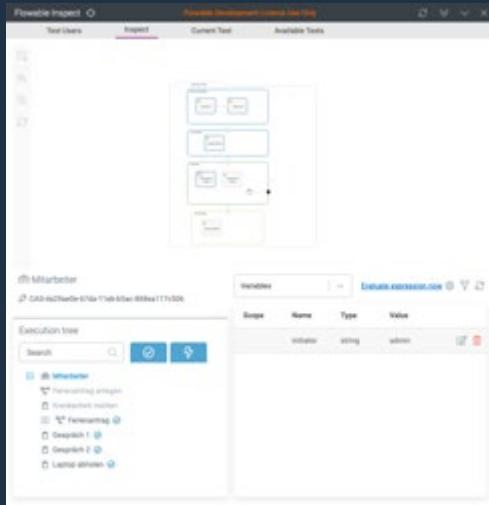
- Enable Flowable Inspect (please don't do this in production!):

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

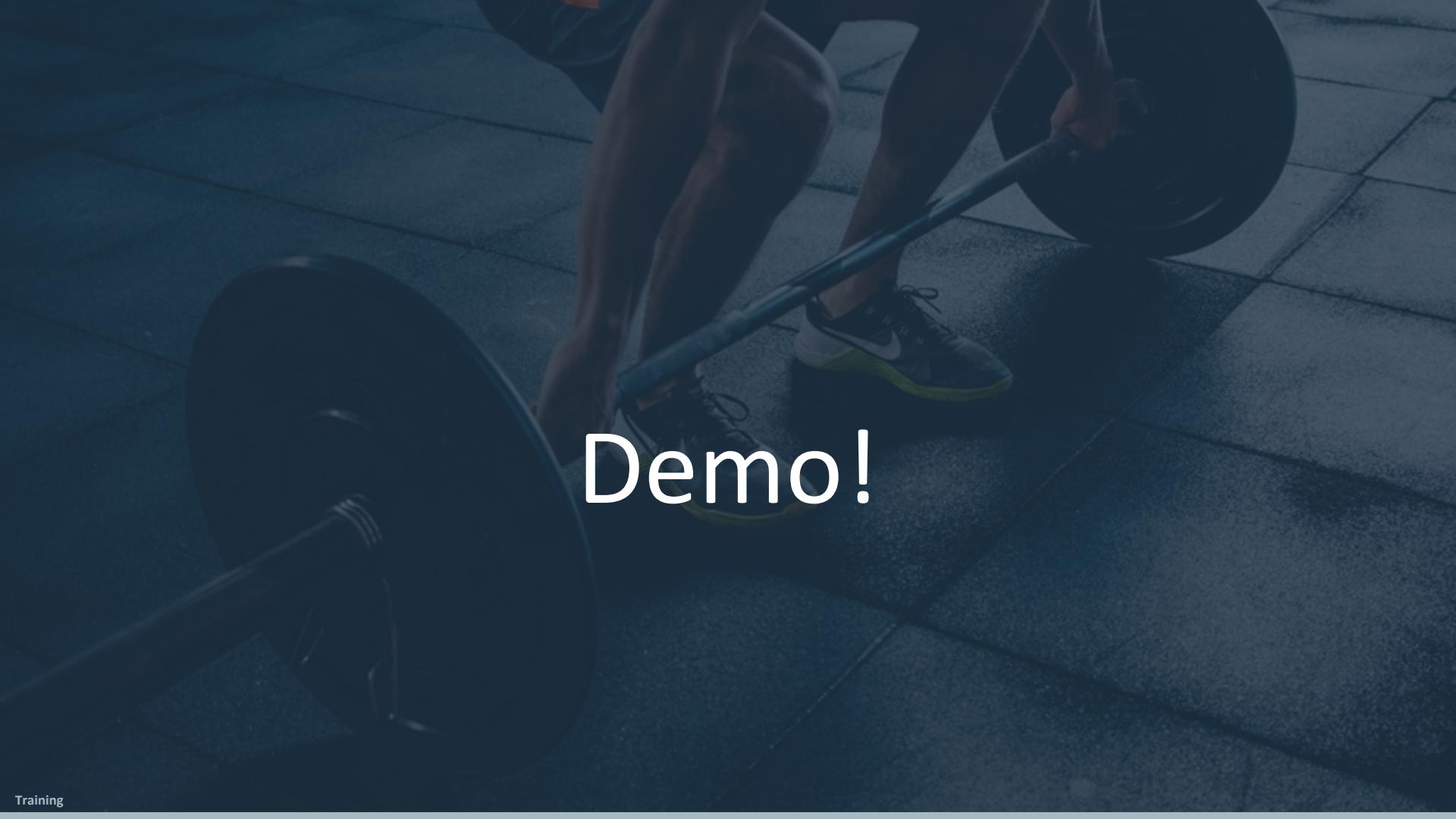
- Click on the profile to enable inspect



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)
-

A person is performing a deadlift with a barbell. The person is wearing a dark t-shirt, light-colored shorts, and grey athletic shoes with white stripes. The barbell has black weight plates. The background is a gym floor with white chalk lines.

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.



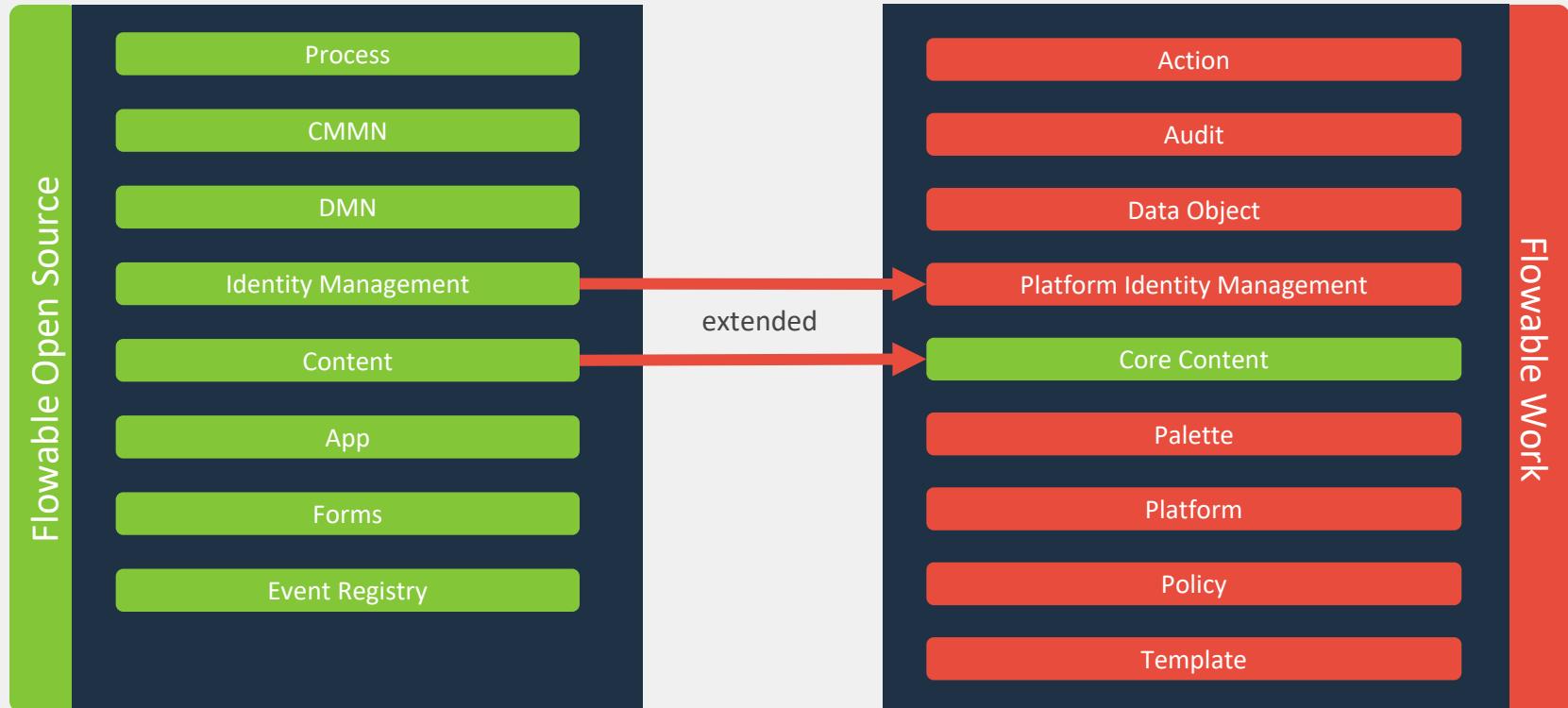
What is an Engine?

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!**



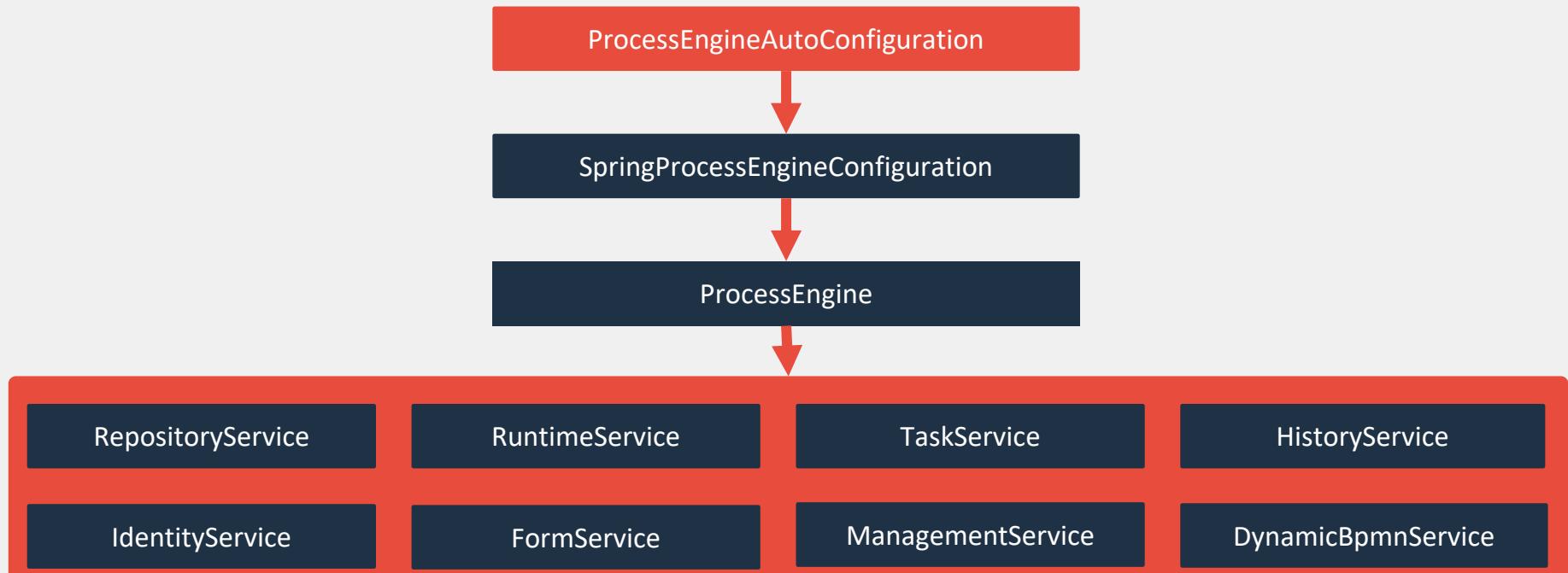
List of Engines



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;

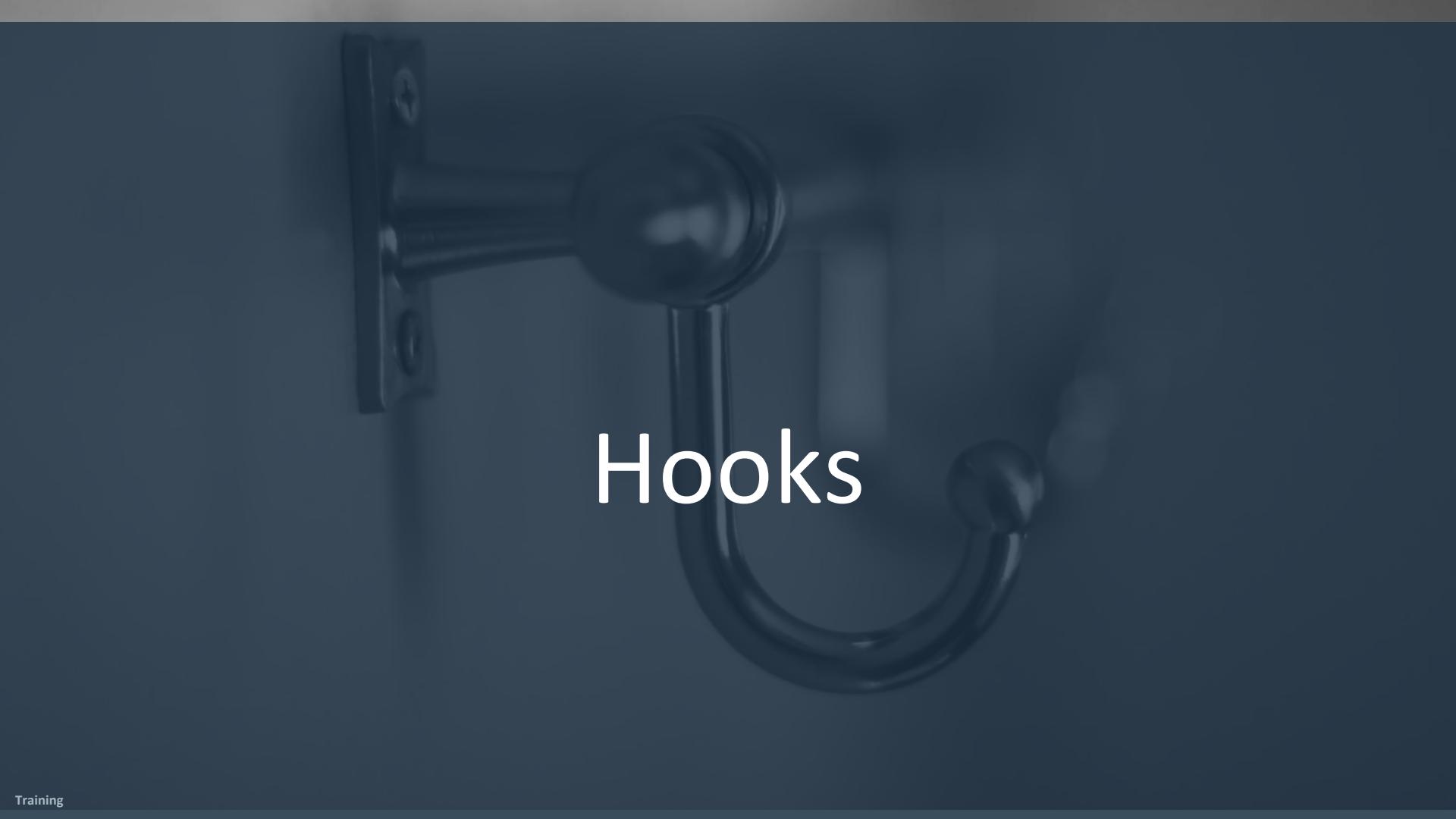
    @Value("${http.connection.socketTimeout:5000}")
    private int socketTimeout;

    @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())); ← Custom Listener

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

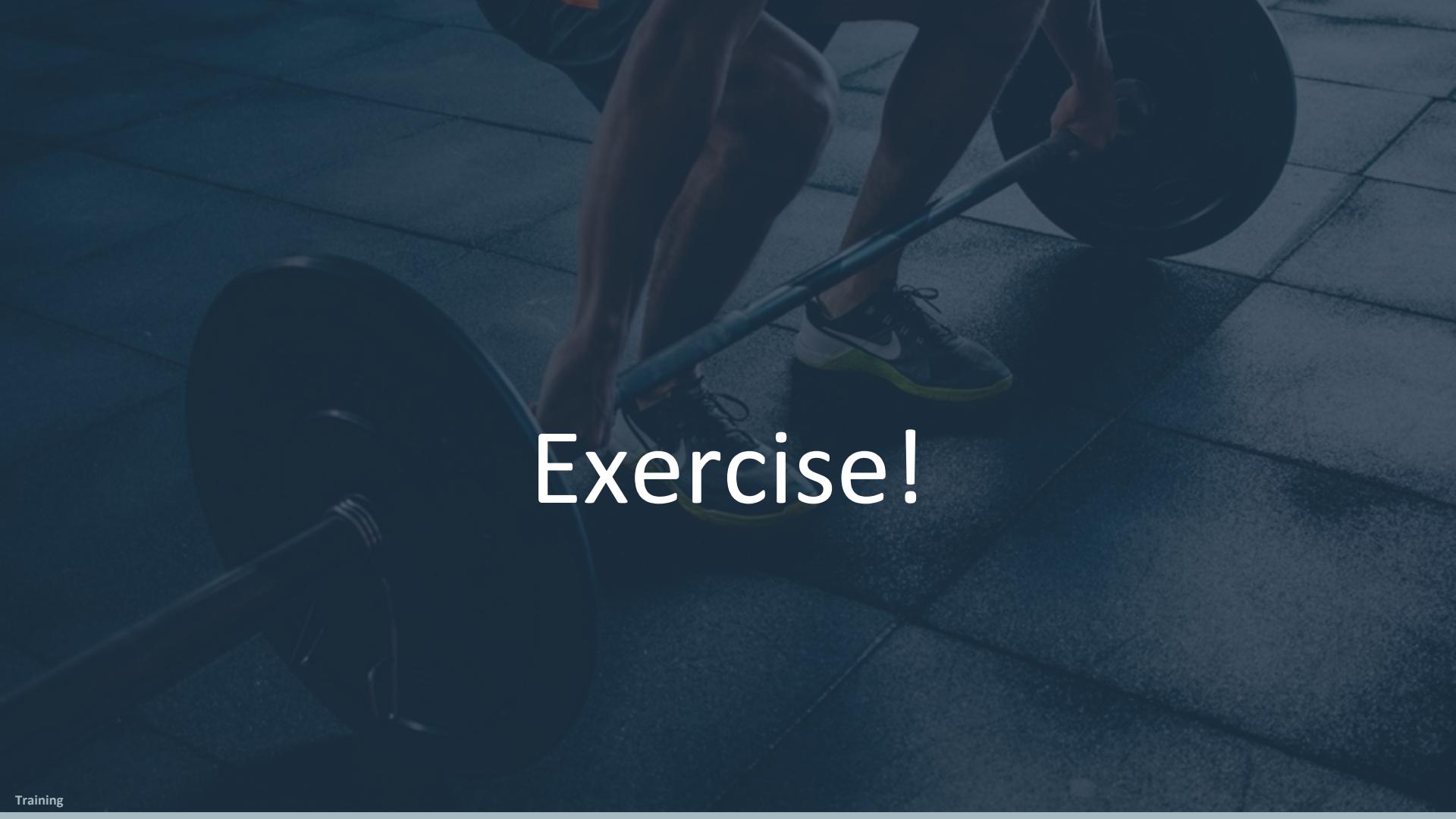
- E.g.:

```
@Configuration  
public class BpmnListenersConfiguration implements  
EngineConfigurationConfigurer<SpringProcessEngineConfiguration>
```

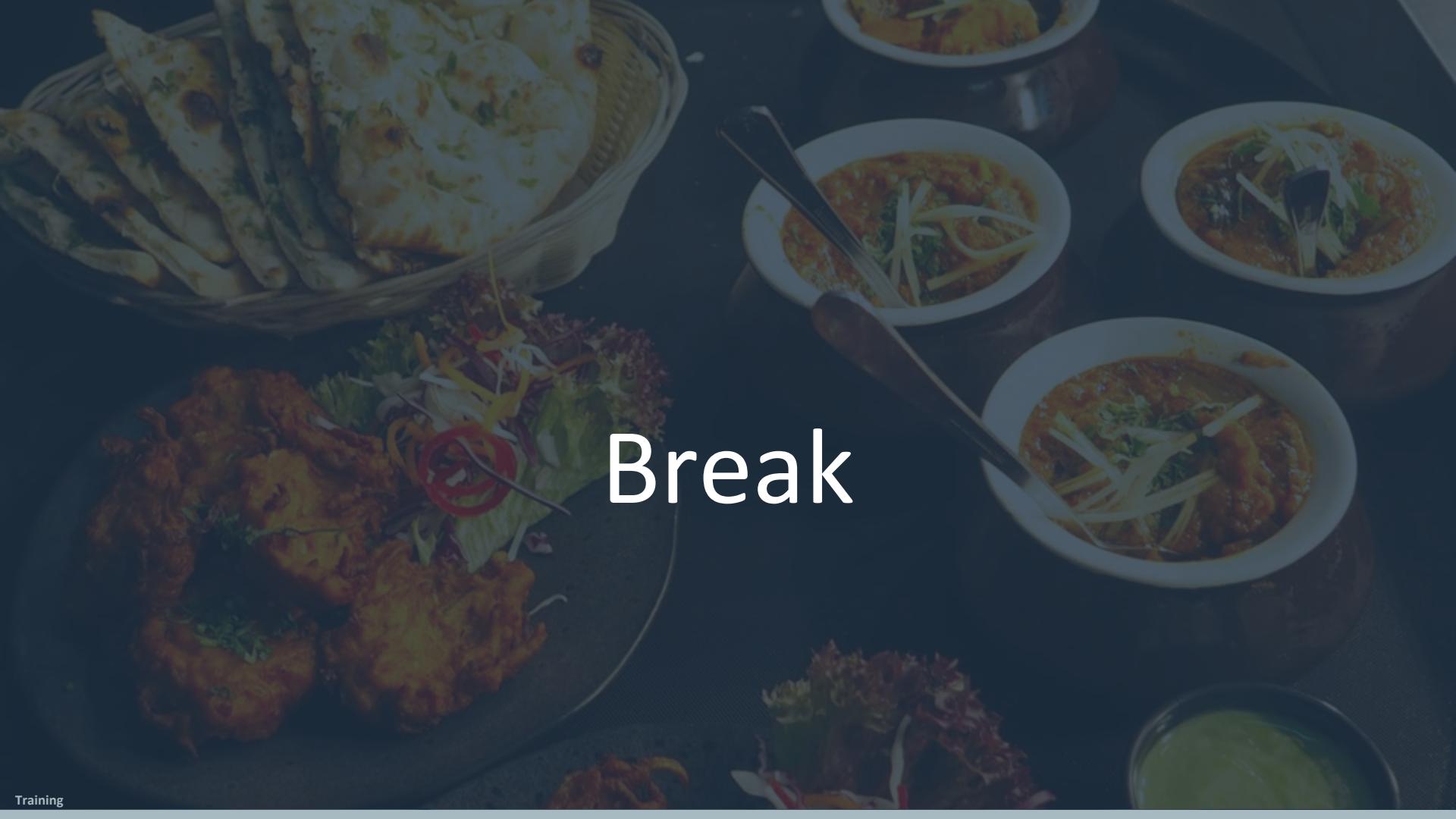
- Add listeners or parse handlers

```
engineConfiguration.setPostBpmnParseHandlers(postBpmnParseHandlers);  
engineConfiguration.setEventListeners(eventListeners);
```

- Listeners inherit from AbstractFlowableEventListener

A person is performing a deadlift with a barbell. The person is wearing a dark t-shirt, dark shorts, and grey athletic shoes with yellow accents. The barbell has two large black weight plates on each side. The background is a gym floor with white chalk lines.

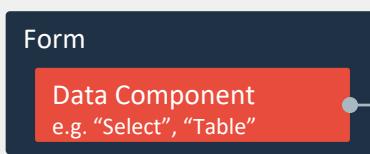
Exercise!

A collage of various Asian dishes, including dumplings, soups, and salads, arranged in a circular pattern. The dishes are presented in different bowls and plates, showcasing a variety of colors and textures.

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



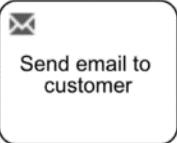
- Call external system from process
- Bind data from endpoint response
- Send data to endpoint



- Develop independent microservice
- Integrate via the Public REST API

zero code

low code



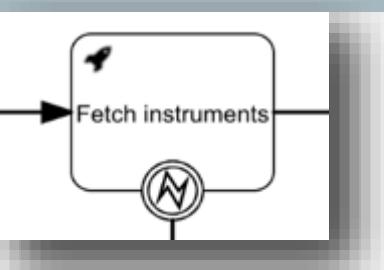
Mail Task

- 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
```

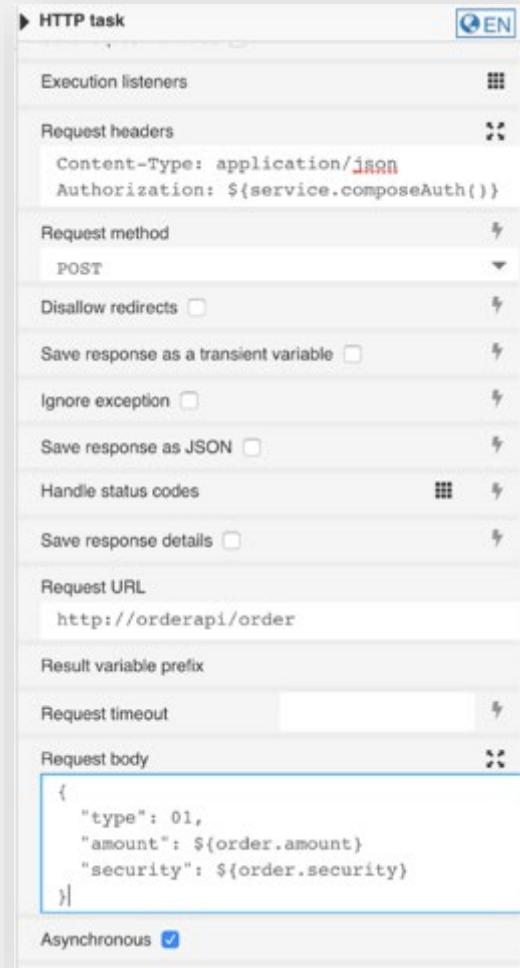
The screenshot shows the 'Details' tab of a Mail Task configuration in the Flowable BPMN editor. The configuration includes:

- Execution listeners**: None
- Email properties**: subject:Order confirmation, from:services@flowbank.io
- Subject**: Order confirmation
- From**: services@flowbank.io
- To**: \${customer.mail}
- Cc**: None
- Bcc**: None
- HTML**: Dear customer, your order \${order.id} has been confirmed!
- Text**: None
- Charset**: None



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**



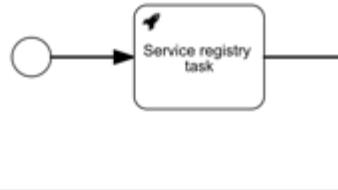
The screenshot shows the configuration interface for an 'HTTP task'. The configuration includes:

- Execution listeners
- Request headers:
 - Content-Type: application/json
 - Authorization: \${service.composeAuth()}
- Request method: POST
- Disallow redirects:
- Save response as a transient variable:
- Ignore exception:
- Save response as JSON:
- Handle status codes
- Save response details:
- Request URL: http://orderapi/order
- Result variable prefix
- Request timeout
- Request body:

```
{  
    "type": "01,  
    "amount": ${order.amount}  
    "security": ${order.security}  
}
```
- Asynchronous:

General Architecture Rule

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

🔍 Enter text to filter attributes ...

General

Model Id	serviceRegistryTa
Name	Service registry
Documentation	

Details

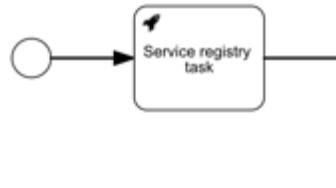
Service model	Hello World S
Service operation	Select operation...
No items matching "" were found.	
Save output variables as transient variable <input type="checkbox"/>	
Output variable	
Exception Mappings	

Execution

Asynchronous <input type="checkbox"/>
Execution listeners
Skip expression
Is for compensation <input type="checkbox"/>

Multi instance

Multi instance type
None
Collection
Element variable



Service Registry

General configuration

A **standard** service model defines operations with input and output parameters. This kind of service model has less freedom to define the service interface.

Type: Standard

Service type: REST

REST Settings

Base URL:

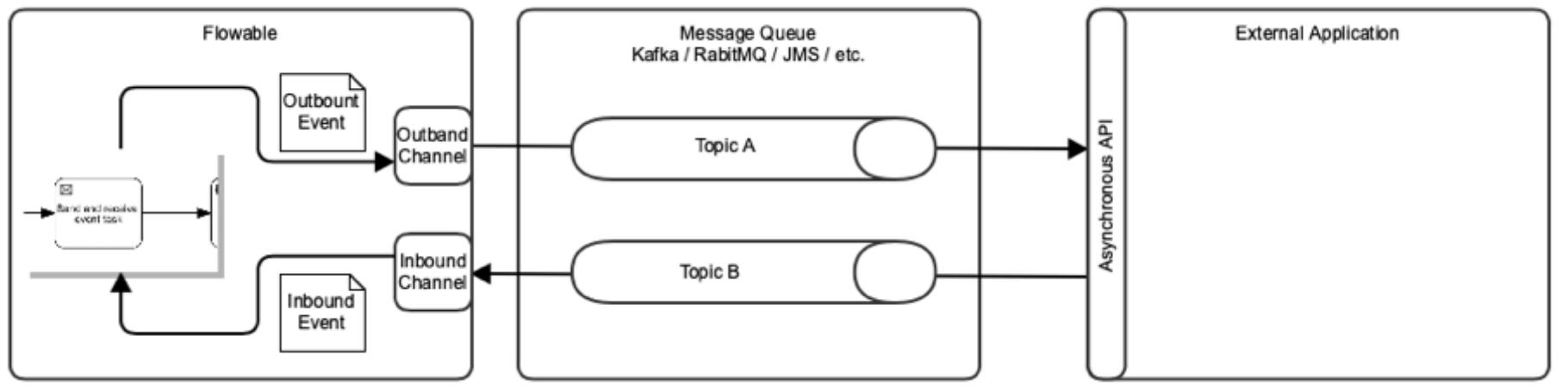
Custom headers:

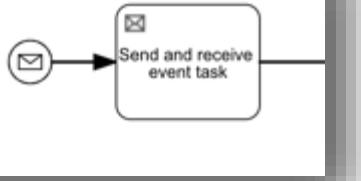
Operations

New operation

Label	<input type="text" value="Operation 1"/>																
Key	<input type="text" value="operation1"/>																
Description	<input type="text"/>																
Input parameters <p>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).</p>																	
REST Configuration <table> <tr> <td>URL <small>(?)</small></td> <td><input type="text"/></td> </tr> <tr> <td>Method <small>(?)</small></td> <td><input type="button" value="▼"/></td> </tr> <tr> <td>Headers <small>(?)</small></td> <td><input type="button" value="+ Add header"/></td> </tr> <tr> <td>Output path <small>(?)</small></td> <td><input type="text"/></td> </tr> </table>		URL <small>(?)</small>	<input type="text"/>	Method <small>(?)</small>	<input type="button" value="▼"/>	Headers <small>(?)</small>	<input type="button" value="+ Add header"/>	Output path <small>(?)</small>	<input type="text"/>								
URL <small>(?)</small>	<input type="text"/>																
Method <small>(?)</small>	<input type="button" value="▼"/>																
Headers <small>(?)</small>	<input type="button" value="+ Add header"/>																
Output path <small>(?)</small>	<input type="text"/>																
Add parameter <table> <tr> <td>Label</td> <td><input type="text" value="Input 1"/></td> </tr> <tr> <td>Name</td> <td><input type="text" value="input1"/></td> </tr> <tr> <td>Type</td> <td><input type="button" value="String"/></td> </tr> <tr> <td>Body location <small>(?)</small></td> <td><input type="text"/></td> </tr> <tr> <td>Excluded from body <small>(?)</small></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Description</td> <td><input type="text"/></td> </tr> <tr> <td>Required</td> <td><input type="checkbox"/></td> </tr> <tr> <td>Default value</td> <td><input type="text"/></td> </tr> </table>		Label	<input type="text" value="Input 1"/>	Name	<input type="text" value="input1"/>	Type	<input type="button" value="String"/>	Body location <small>(?)</small>	<input type="text"/>	Excluded from body <small>(?)</small>	<input type="checkbox"/>	Description	<input type="text"/>	Required	<input type="checkbox"/>	Default value	<input type="text"/>
Label	<input type="text" value="Input 1"/>																
Name	<input type="text" value="input1"/>																
Type	<input type="button" value="String"/>																
Body location <small>(?)</small>	<input type="text"/>																
Excluded from body <small>(?)</small>	<input type="checkbox"/>																
Description	<input type="text"/>																
Required	<input type="checkbox"/>																
Default value	<input type="text"/>																

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

Send and receive event task

EN

Enter text to filter attributes ...

General

Model Id	sendAndReceiveEventTask1
Name	Send and receive event task
Documentation	

Execution

Outbound event	Demo Outbound Event
Configured to send to	'demoOutboundEvent'
Outbound channel	Kafka Outbound Channel
Send on internal channel	<input type="checkbox"/>
Inbound event	
Inbound channel	Kafka Inbound Channel
Asynchronous	<input type="checkbox"/>
Execution listeners	
Skip expression	
Is for compensation	<input type="checkbox"/>

Multi instance

Multi instance type	None
Collection	
Element variable	
Element index variable	IndexVariable

Event Registry

Event

Event Configuration

Channel

Event Registry

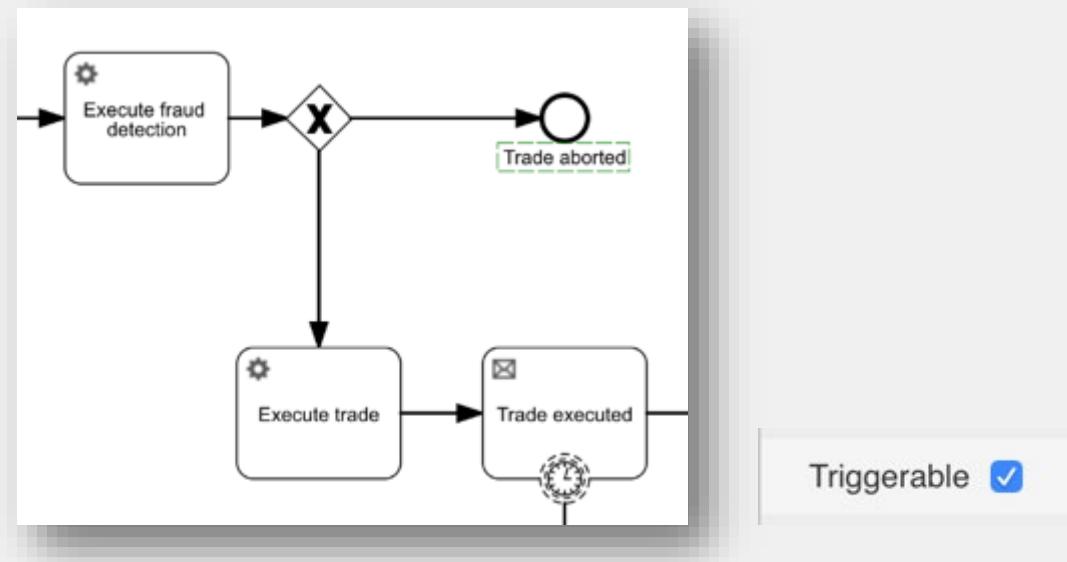
Training

© 2021 Flowable AG. All rights reserved.

Triggerable Service Task

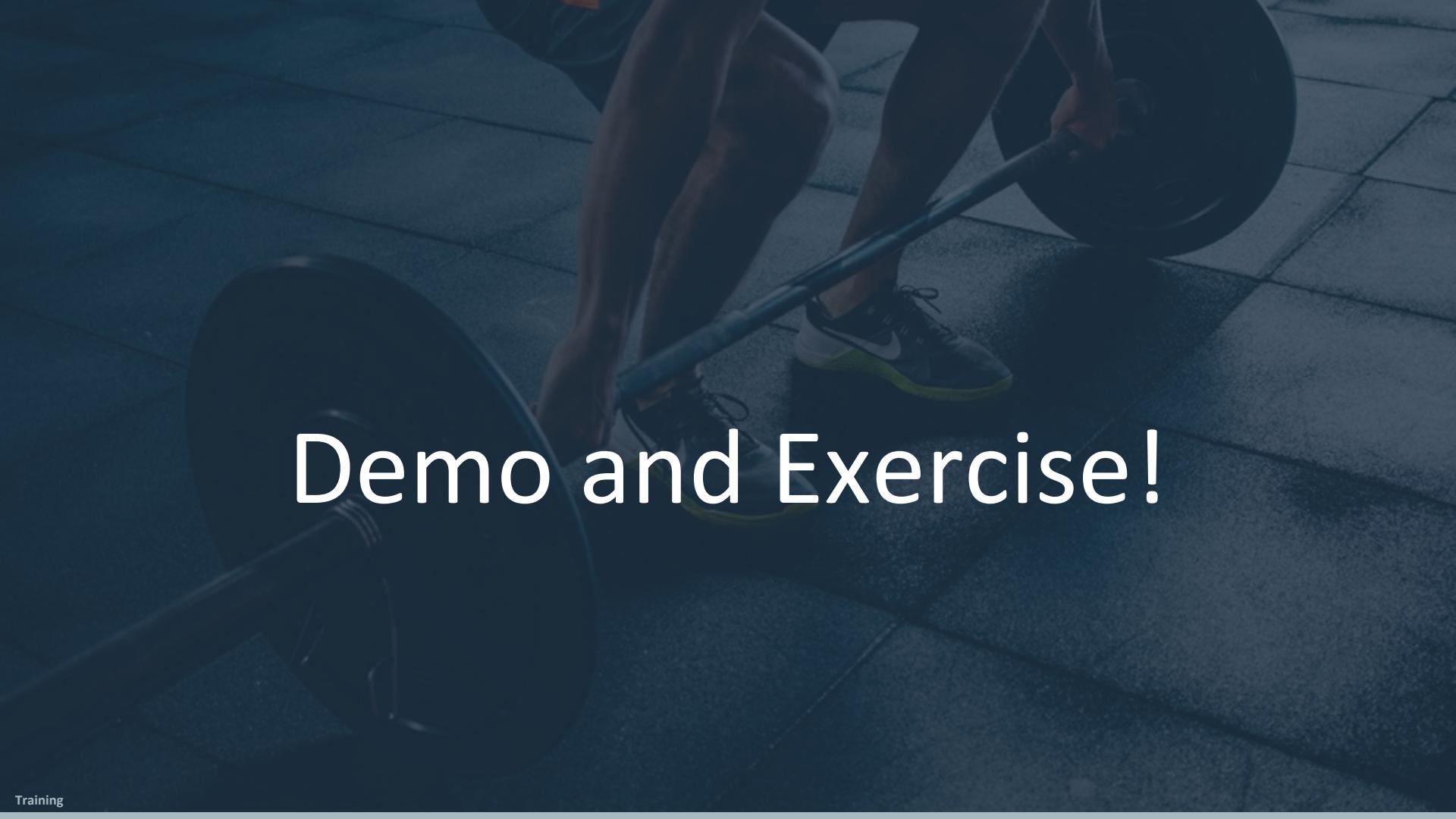
Use the ‘triggerable’ property on the service task for potentially **Long Running Service Tasks**.

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



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

Triggerable

A person is performing a deadlift with a barbell. The person is wearing a dark t-shirt, dark shorts, and grey athletic shoes with yellow accents. The barbell has two large black weight plates on each side. The background is a gym floor with white chalk lines.

Demo and Exercise!

Operations

Running Flowable Platform in different Scenarios

Architecture

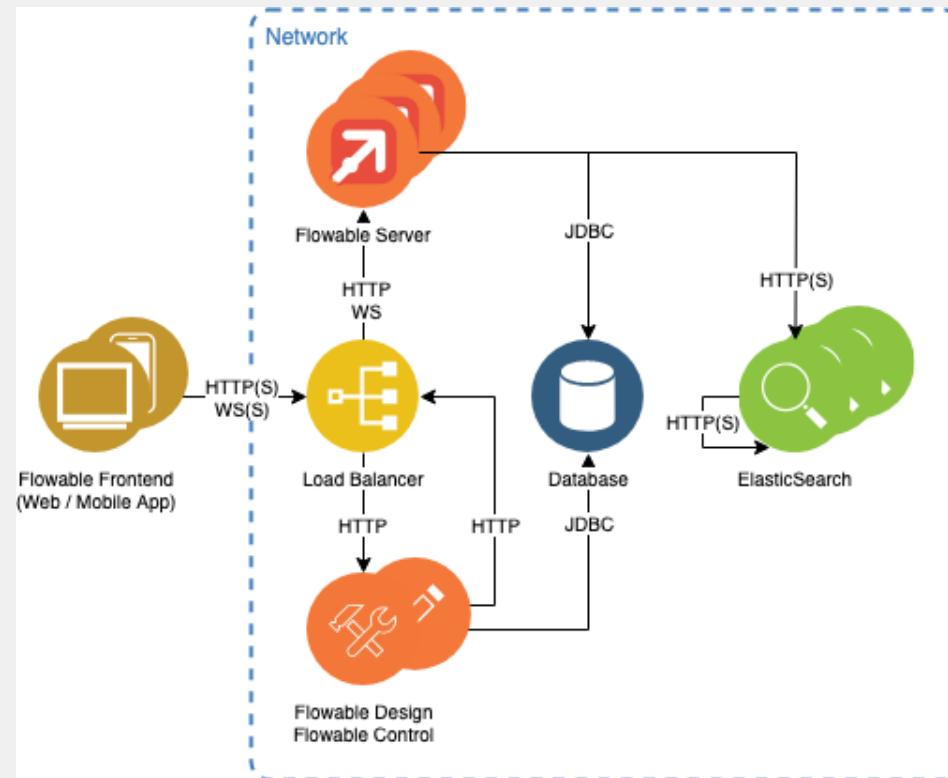
Flowable Server

- 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

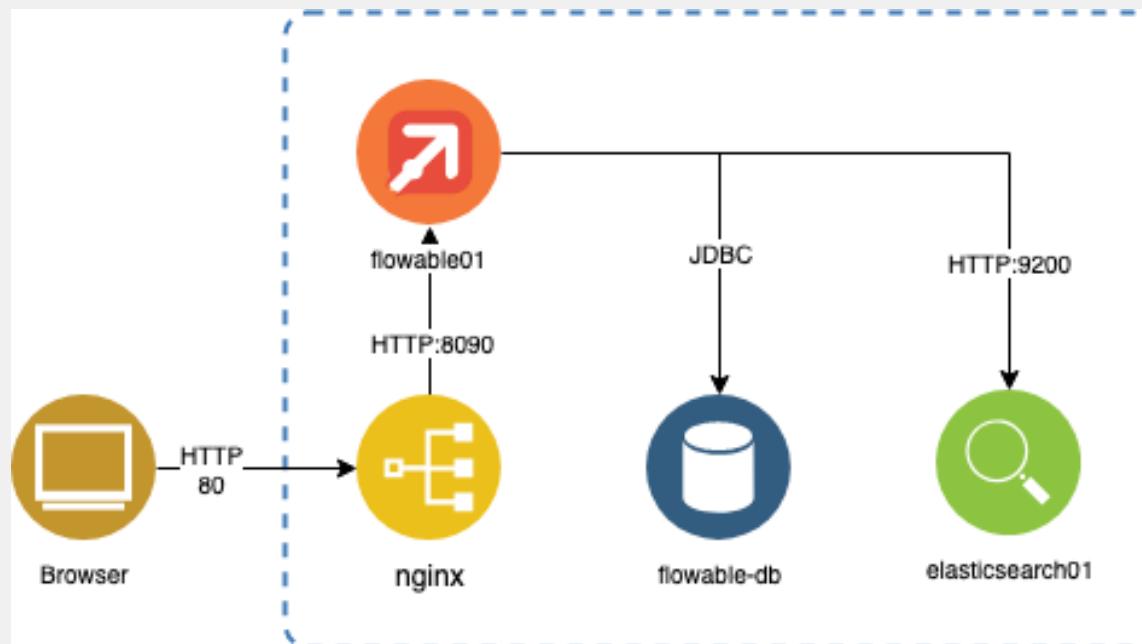
- Chrome, Firefox, Internet Explorer and Safari are supported



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

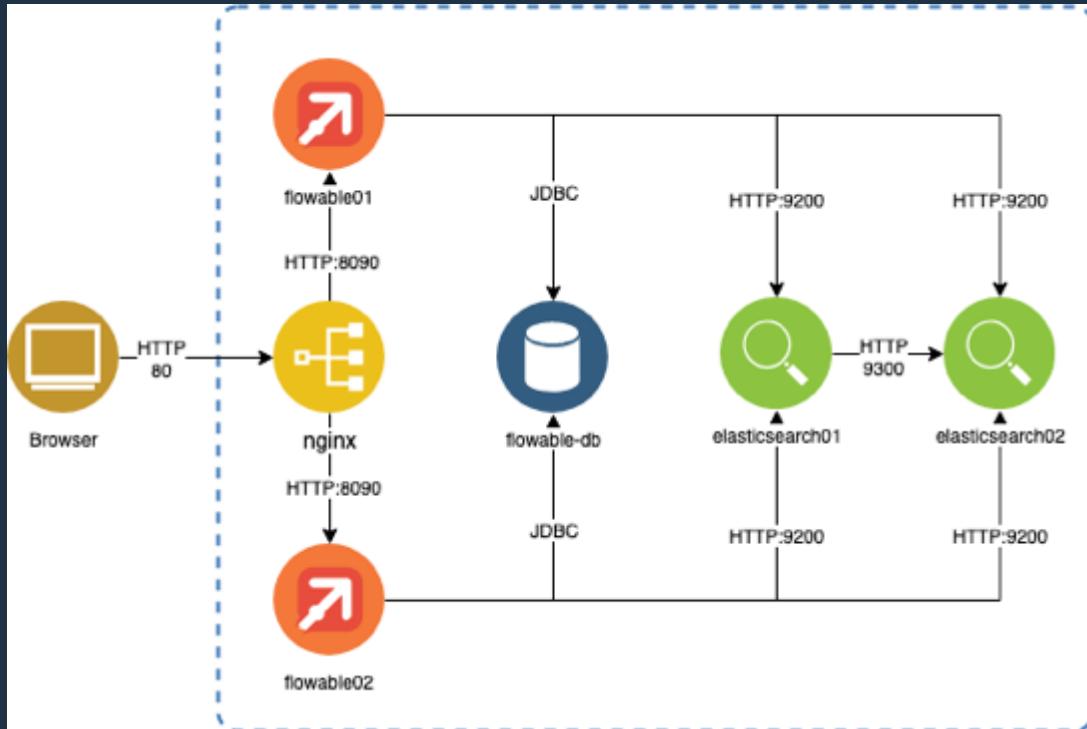
Running a single node



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

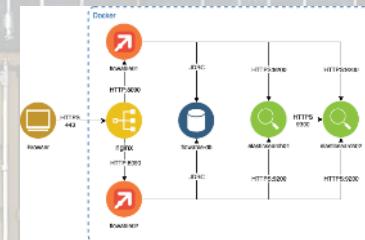
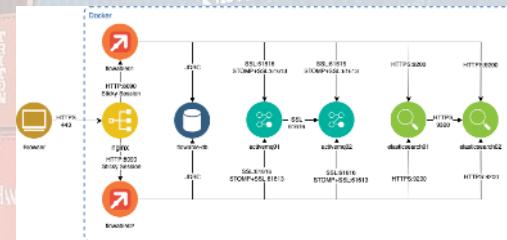
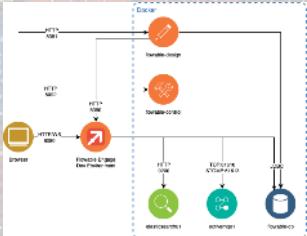


Development setup

- 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

A word on different API methods...



	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 GET /actuator/info

```
{  
    "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",  
                [...]  
            }  
        }  
    }  
}  
{  
    "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?)

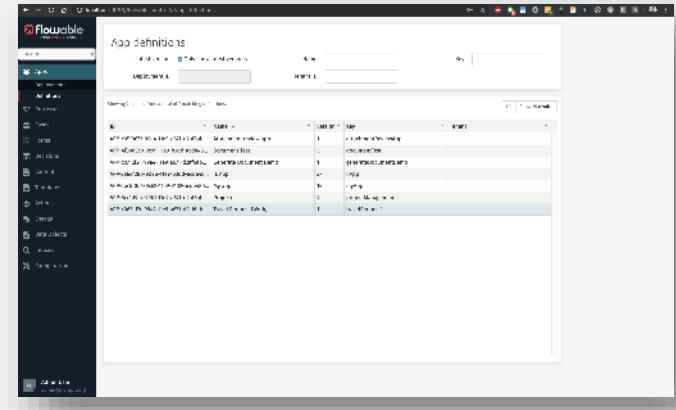




Flowable Control

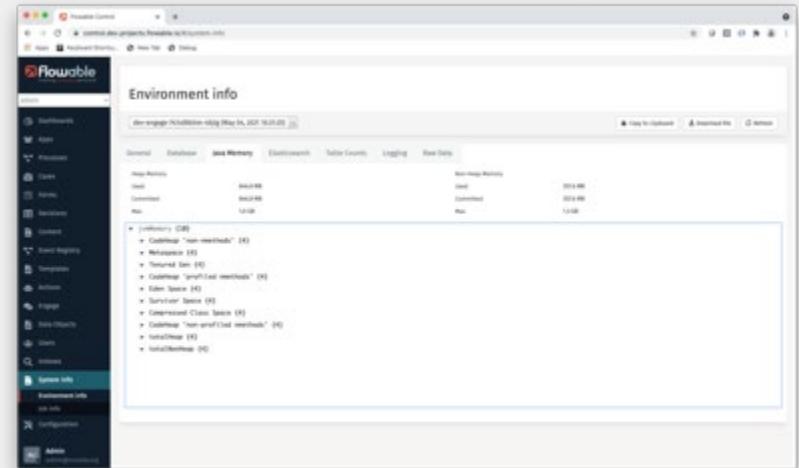
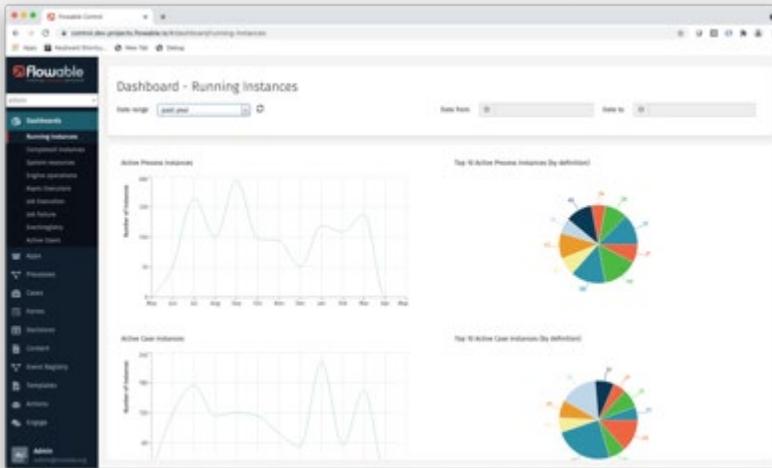
Overview

- 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

The screenshot shows the Flowable Control web application. On the left is a sidebar with navigation links: Default Cluster, Apps, Deployments (which is selected), Definitions, Processes, Cases, Forms, Decisions, Content, Templates, Actions, and Engage. The main content area has a title 'Deployments' and two search input fields: 'Name' and 'Tenant identifier'. Below these is a message: 'Showing 25 results, from a total of 318 matching deployments.' A table lists the deployments with columns: ID, Name, Deploy time, Category, and Tenant. The table contains 25 rows of deployment data.

ID	Name	Deploy time	Category	Tenant
APP-d6802480-2abe-11e9-9ae6-0242...	Attachmentreviewapp	07-02-2019 10:57:57		
APP-d88d43ce-2abc-11e9-9ae6-0242...	GeneralDocumentDemo	07-02-2019 10:57:57		
APP-076f7951-2ac0-11e9-9ae6-0242...	EmployeeLifecycleApp.zip	07-02-2019 11:06:26		
APP-3c4300a9-2acd-11e9-9ae6-0242...	loanapp.zip	07-02-2019 11:06:33		
APP-bc713112-ac01-11e9-9ae6-0242...	recruitingApp.zip	07-02-2019 11:11:31		
APP-67a2c089-2acc-11e9-9ae6-0242...	recruitingApp.zip	07-02-2019 16:24:05		
APP-7fbfd253-2acc-11e9-9ae6-0242...	recruitingApp.zip	07-02-2019 16:24:39		
APP-3ebfd129-2ba4-11e9-afb4-0242...	loanapp.zip	08-02-2019 14:20:03		

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

The screenshot shows the Flowable Control interface with the 'Processes' tab selected in the sidebar. The main area displays a table of process definitions. The table has columns for ID, Name, Version, Key, and Tenant. The data includes:

ID	Name	Version	Key	Tenant
PRC-agendaItem:158b2be8f2edd-11e9-...	Agenda Item	1	agendaItem	
PRC-agendaPublish:158b3d000-2edd-11e9-...	Agenda Publish	1	agendaPublish	
PRC-contentModelProcess:15105c14-9...	Content Model Process	1	contentModelProcess	
PRC-dateFormProcess:134994a29d-a24f-...	Date Form Process	13	dateFormProcess	
PRC-endEventProcess:107e9015-2ac0-11...	End Event Process	6	endEventProcess	
PRC-ExpenseReport:07e9015-2ac0-11...	Expense Report	1	ExpenseReport	
e637cf4f-7287-11e9-9fb2-0242a2120000	Expenses Report Approval Process	65	expensesReportApprovalProcess	
PRC-FAQ:T1:4:/00c9cce-1fb-11e9-b0eb-0...	FAQ-T1	4	FAQ-T1	

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

The screenshot shows the Flowable process instance management interface. On the left is a sidebar with navigation links: Localhost, Apps, Processes (selected), Deployments, Definitions, Instances (selected), Tasks, Jobs, Event subscriptions, Cases, Forms, Decisions, Content, and Templates. The main area has a title 'Process instances' and several search filters: Process definition (All process definitions), Status (Active instances), Superinstance ID, Started after, Ended after, Started before, Ended before, and Business key. Below the filters, a message says 'Showing 10 results, from a total of 10 matching process instances.' A 'Show 25 results' link is at the bottom right. A table lists 10 process instances with columns: ID, Business key, Process definition, Created, and Ended. The table includes rows for PRC-5115659f-a30f-11e9-8a5c-acde48001122, PRC-6fb7ab6e-a30c-11e9-8a5f-acde480011..., PRC-1d9993a7-e30c-11e9-8a5f-acde480011..., and PRC-56cb8169-a2f7-11e9-af13-acde48001122, all associated with 'Some Process' and 'Balsamico' respectively, and created on 10-07-2019.

ID	Business key	Process definition	Created	Ended
PRC-5115659f-a30f-11e9-8a5c-acde48001122		Some Process	10-07-2019 14:36:19	
PRC-6fb7ab6e-a30c-11e9-8a5f-acde480011...		Some Process	10-07-2019 14:35:42	
PRC-1d9993a7-e30c-11e9-8a5f-acde480011...		Some Process	10-07-2019 14:12:24	
PRC-56cb8169-a2f7-11e9-af13-acde48001122		Balsamico	10-07-2019 11:44:40	



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

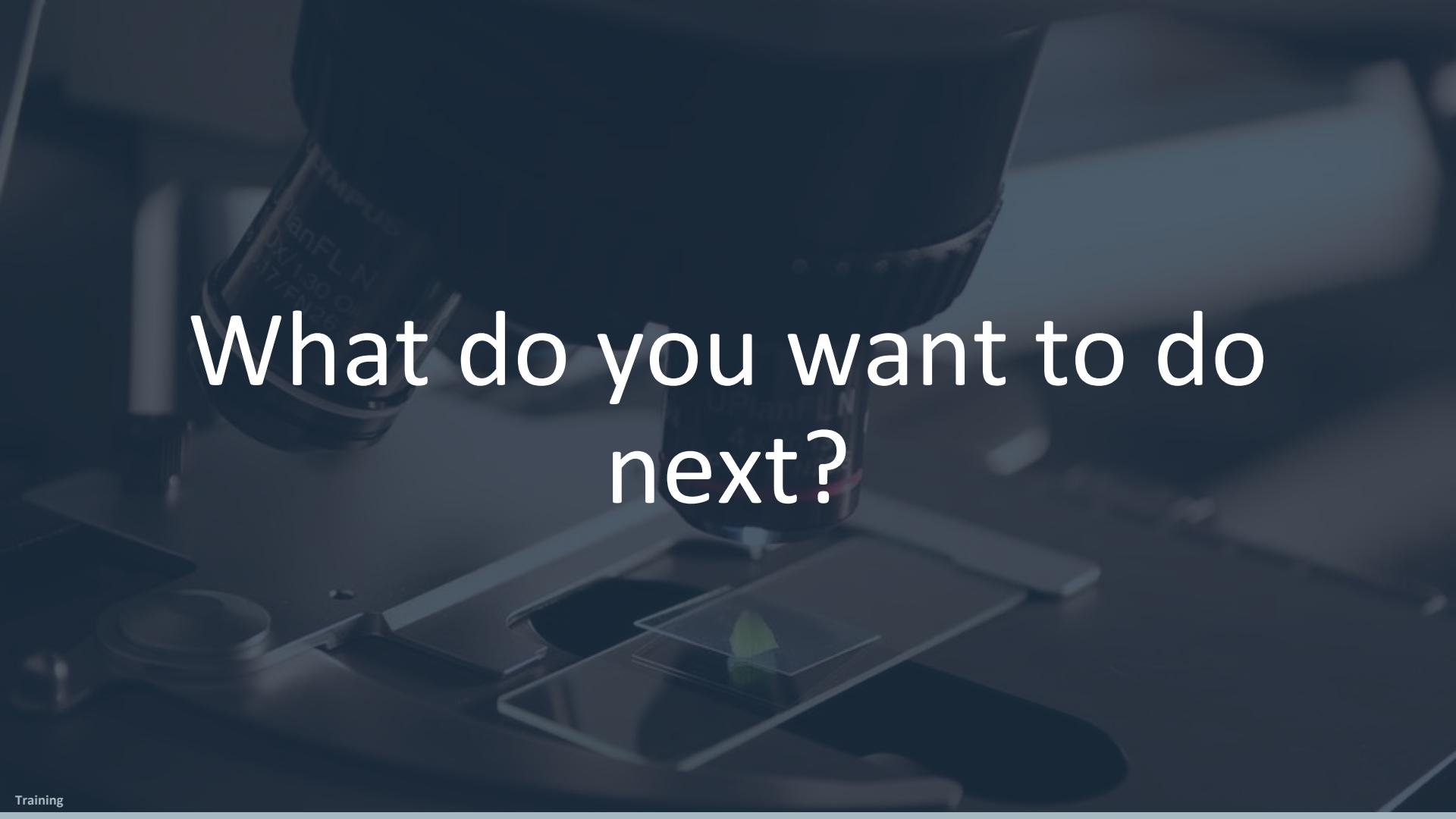
- Set up **multiple clusters** (=environments)
- Possibility to individually configer each API separately

- Configuration:

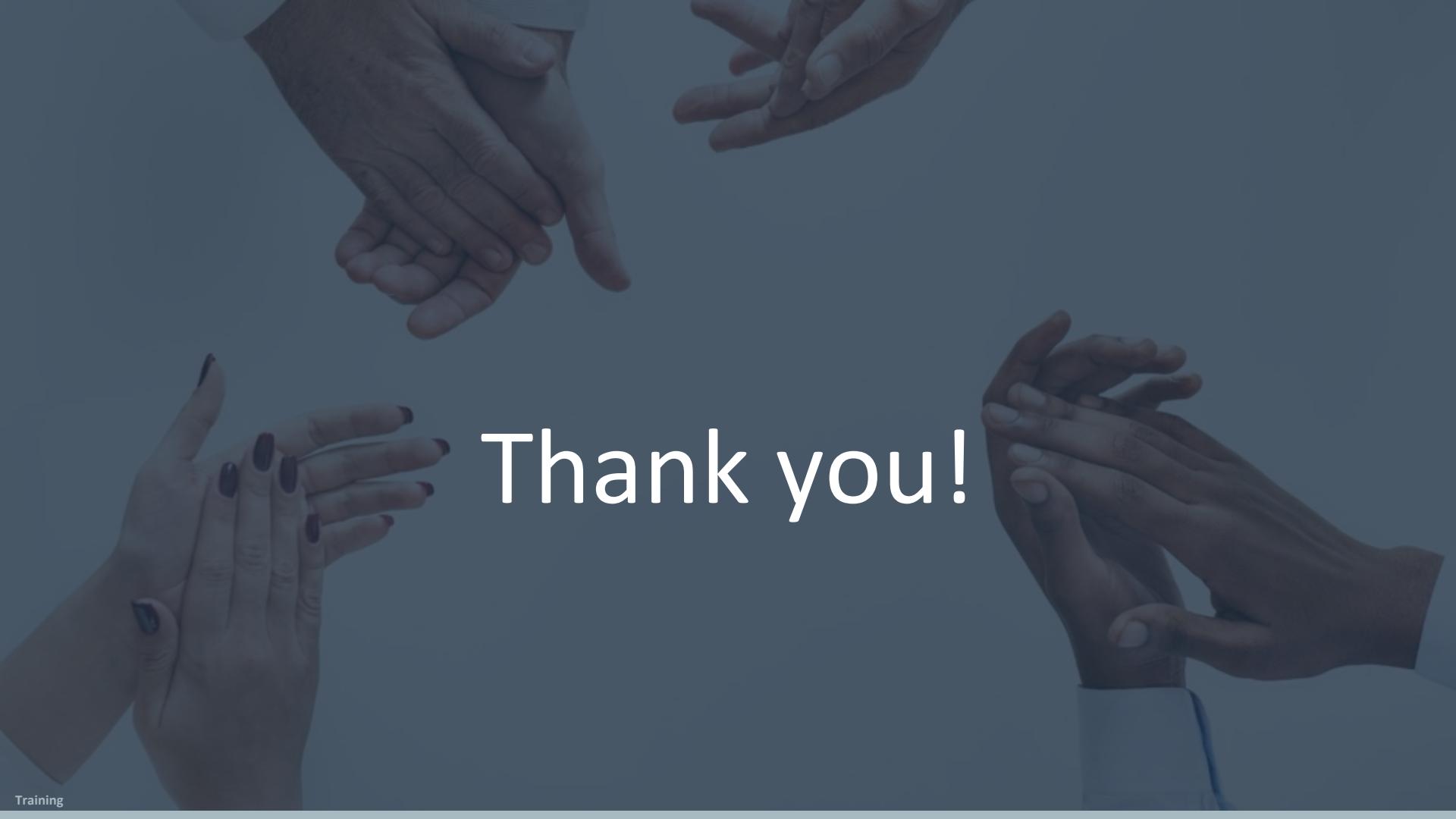
- Name
- Server Address
- Context Root
- Port
- User name and password

The screenshot shows the Flowable Platform interface. On the left is a dark sidebar menu with items like Apps, Processes, Cases, Forms, Decisions, Content, Templates, Actions, Engage, Data Objects, and Indexes. Below these are Configuration, Clusters, User management, and Change password. The Configuration item is currently selected. To the right of the sidebar is a main content area titled "Default Cluster". A sub-header says "This section allows you to define a master configuration for the cluster. All engine endpoints will use these values by default, but can be overridden." Below this are tabs for Details, App, Process, Case, Decision, Form, Content, Action, Engage, Platform, Platform IDM, and Template. The Details tab is active. It displays a table with columns for Name, Description, Server address, Server port, and Username. The data row shows "Name: Default Cluster", "Description: Default Flowable Cluster", "Server address: http://localhost", "Server port: 8080", and "Username: admin". There are also "Edit cluster configuration" and "Delete cluster configuration" buttons at the top of the configuration table.

Name	Description	Server address	Server port	Username
Default Cluster	Default Flowable Cluster	http://localhost	8080	admin

A close-up photograph of a light microscope's eyepiece and objective lenses. A glass slide with a small, green, irregularly shaped sample is positioned on the stage. The background is dark.

What do you want to do
next?

A photograph showing several pairs of hands clapping against a dark, slightly blurred background. The hands are of different skin tones and are captured in various stages of a clap. In the center, the words "Thank you!" are overlaid in a large, white, sans-serif font.

Training

Thank you!

