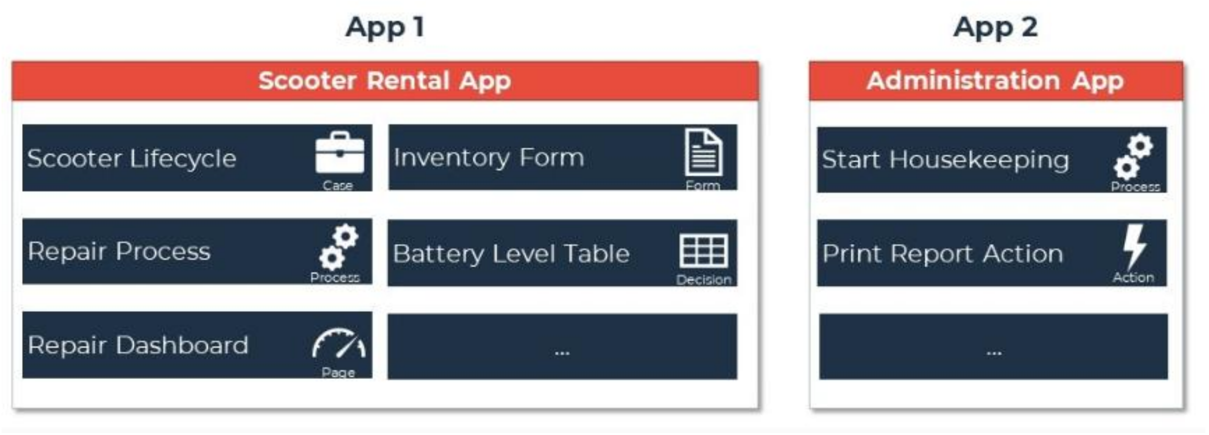
Flowable Development Essentials

Versioning, Deployment and Apps

In the previous chapter, we explored the "Model to Instance" lifecycle in Flowable. The key mechanism enabling this transformation is called **deployments**. This chapter explores how deployments work, the role of versioning, and their management in production environments.

How Deployment Works

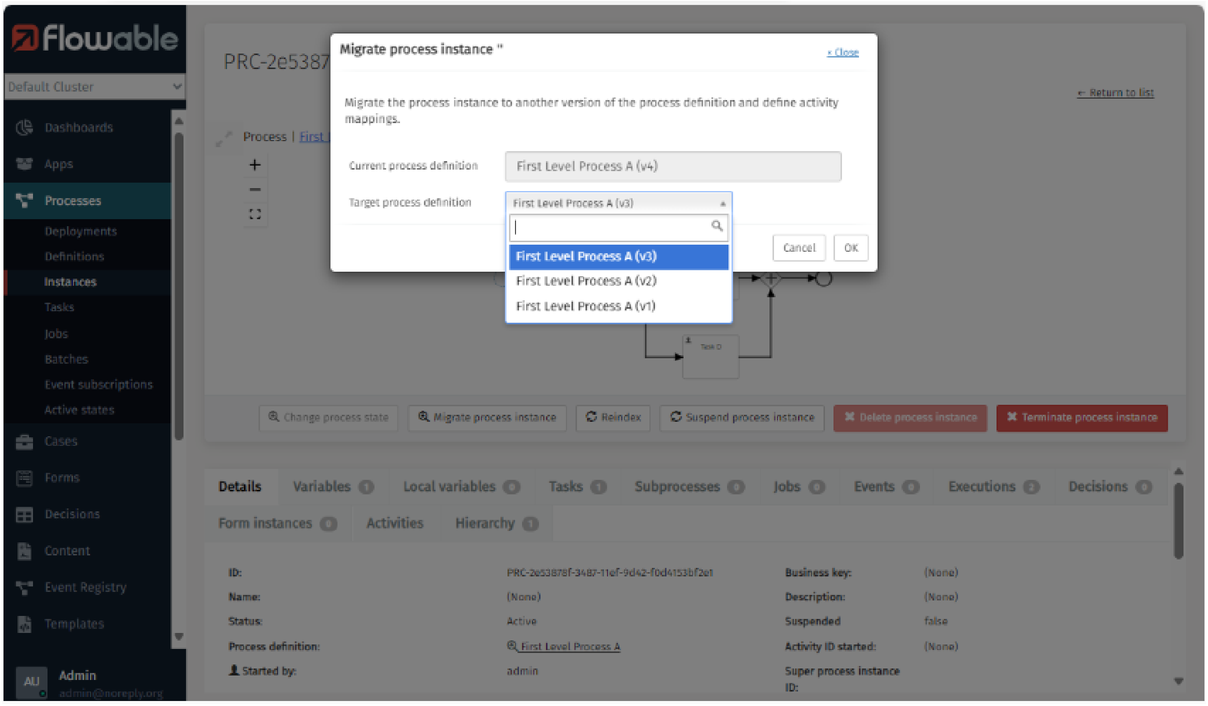
In Flowable, models—whether they represent processes, cases, forms, or other components—are organized into **Apps** within Flowable Design. An App serves as a container that groups related models for deployment.



When you **publish** an App from Flowable Design, a new deployment begins. Here's what happens:

1. **Packaging**: All models within the App are zipped into a package (BAR file). This package contains XML or JSON representations according to the type of model.
2. **Transfer to Flowable Work**: The zipped package is sent via REST to Flowable Work.
3. **Unpacking**: Flowable Work unpacks the package and creates a new **App Deployment Entity**. In addition, it generates individual deployments for each model type, such as Process Deployment and Form Deployment.
4. **New definitions** (or new versions of existing definitions) are created.

Managing Instances and Migrations



Migrate process instance ''

× Close

Migrate the process instance to another version of the process definition and define activity mappings.

Current process definition	First Level Process A (v4)
Target process definition	First Level Process A (v3)

First Level Process A (v3)
First Level Process A (v2)
First Level Process A (v1)

Cancel OK

ID:	PRC-2e53878f-3487-11ef-9d42-f0d4153bf2e1	Business key:	(None)
Name:	(None)	Description:	(None)
Status:	Active	Suspended	false
Process definition:	First Level Process A	Activity ID started:	(None)
Started by:	admin	Super process instance ID:	

Existing instances continue to operate on the version of the model they were created with. This ensures stability and consistency. However, instances can be migrated to newer or older versions as requirements evolve:

- **Forward Migration** updates instances to a newer version.
- **Backward Migration** reverts instances to an older version.

Process and Case Migrations are often more complex and involve detailed mappings for activities, variables, and other components, particularly for processes and cases.

Deployment in Production
Deploying models to a production environment is typically handled through code. The process is as follows:

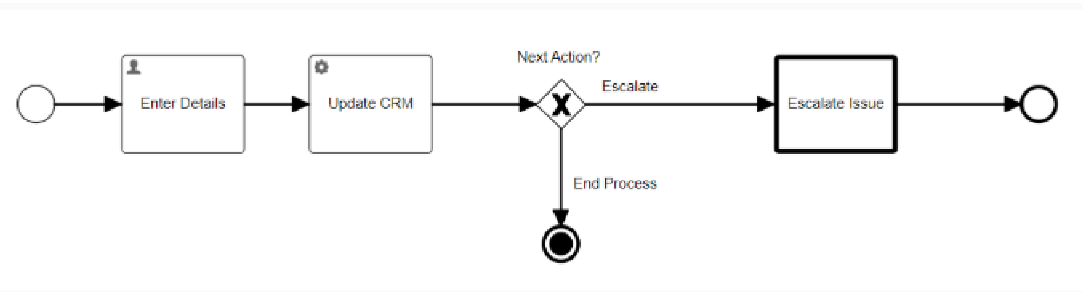
1. **Export App as BAR File**: The App is exported as a BAR file, which is essentially a zipped package of the models. It's the same artifact that is also created when publishing from Flowable Design.
2. **Include in Source Code**: This BAR file is then included in the resources of the project's source code, typically in a folder called "apps".
3. **Start up Deployment**: Upon application start up, Flowable automatically deploys the App from the BAR file. The file is unpacked, and new deployments are created for each model type within the App. By default, apps will only be deployed if there are no newer deployments.

User/Human Tasks

Interact with end users
In Flowable, **Human Tasks** (CMMN) and **User Tasks** (BPMN) allow processes and cases to wait for user input through forms. This input is stored as variables and used to control the workflow or interact with external systems.

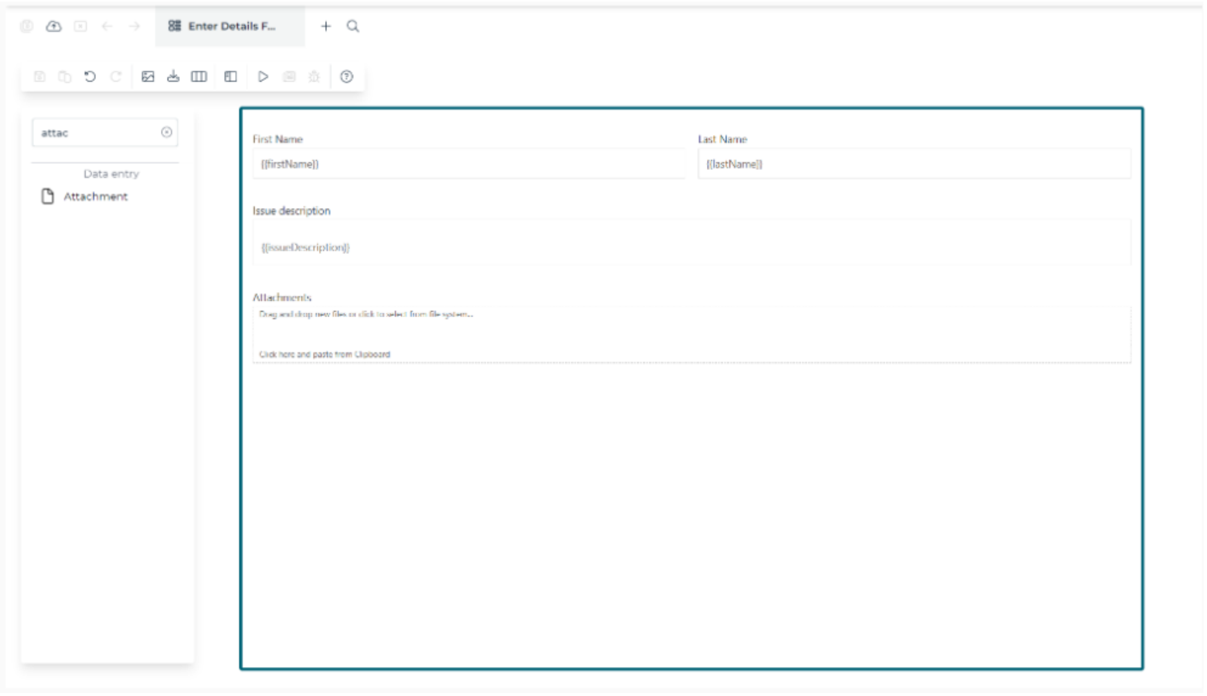
- **Forms and Data**: Users fill out forms linked to these tasks, providing data that is stored in the process or case. This data can control the workflow and be used by other tasks and systems.
- **Assignment**: Tasks can be assigned to users or groups, ensuring the right person handles the task.
- **SLAs and Deadlines**: Tasks can have SLAs, specifying actions if the task is not completed on time, such as sending reminders or escalating the task.

Example
In a customer service workflow, a task might require an agent to fill out a form with issue details.



The entered data could:

- **Control Flow**: Direct the process based on the issue type.
- **External Interaction**: Update a support system.
- **Deadline Handling**: Escalate if unresolved within the SLA timeframe.



First Name
{{firstName}}
Last Name
{{lastName}}
Issue description
{{issueDescription}}
Attachments
Drag and drop new files or click to select from file system...
Click here and paste from Clipboard

Variables in Flowable

Variables are fundamental in Flowable, pivotal for decision-making, integrations, and data handling within processes and cases. They are dynamic entities with names, types, values, and scopes, supporting integrations, calculations, and storing intermediate results.

Key Aspects of Variables

A variable includes:

- **Name**: Identifier for the variable.
- **Type**: Data type (e.g., string, integer, date).
- **Value**: Actual data stored in the variable.
- **Scope**: Context where the variable is available, typically associated with entire process or case instances. Variables can also be stored locally on tasks or other entities in specific scenarios.

Variable Types and Java Integration

Variables can be of types such as:

- **String**: Accessible as `java.lang.String`.
- **Integer**: Accessible as `java.lang.Integer`.
- **Date**: Accessible as `java.util.Date`.
- **JSON**: Accessible as `com.fasterxml.jackson.databind.ObjectNode`.

Accessing Variables in Models

To access variables:

- **Process and Case Models**: Use **${variableName}** syntax to reference variables directly within models, like ${loanAmount}.
- **Forms and Pages**: Use **{{variableName}}** syntax within forms of User Tasks or Human Tasks to bind form fields to variable values, such as {{applicantName}}. The same syntax is used for Pages

Variable Lifecycle

Variables are set or modified at various points:

- **User/Human Tasks**: Forms collect data and set variables.
- **Service Tasks**: Logic in these tasks can create or update variables.
- **Initialize Variables Task**: Special task to set and update variables in processes and cases.

Implicit Data Model

- **Data Dictionary**: Flowable's Data Dictionary models offer structured data management, defining and shaping data explicitly to standardize and document variables used in processes and cases.
- **Dynamic Creation**: Variables can be dynamically created during process or case execution without predefined definitions.

Example
In a loan approval process:

- **User Input**: Forms collect loan details, setting variables like loan amount and applicant details.
- **Service Task Processing**: Tasks calculate eligibility scores and update variables accordingly.
- **Decision Points**: Variables drive flow decisions, determining loan approval or rejection.

Variables from the foundation of Flowable applications, enabling dynamic workflows, adaptable data handling, and seamless integration with Java code. They evolve as processes or cases progress, with the option to define them more formally using Data Dictionary models when needed.

Flowable Project Setup

Setting up your project

To begin developing with Flowable Work on your local machine, follow these instructions:

1. **Prerequisites:**
   - If participating in a training, you will receive a Flowable commercial license or development license along with repository access.

    - Ensure you have a Java development environment (Java 17 or newer). We recommend IntelliJ IDEA Ultimate as your IDE.
2. **Copy licenense into user folder:**
    - Ensure your license file is placed in the .flowable directory in your user folder: `C:\Users\YourUser\.flowable\flowable.license` for Windows and `/Users/YourUser/.flowable/flowable.license` for MacOS.
3. **Configure Maven:**
    - Add Flowable Artifactory to your Maven `settings.xml` file (replace placeholders with your credentials):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <servers>
        <server>
            <id>flowable-mirror</id>
            <username>your-artifact-user-account</username>
            <password>xxxxxxxxxxxxxxxx</password>
        </server>
    </servers>
    <profiles>
        <profile>
            <id>flowable-artifacts</id>
            <repositories>
                <repository>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                    <id>flowable-mirror</id>
                    <name>flowable-maven-all</name>

<url>https://artifacts.flowable.com/artifactory/flowable-maven-
all</url>;
                </repository>
                <repository>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                    <id>central</id>
                    <name>central</name>
                    <url>https://repo1.maven.org/maven2</url>;
                </repository>
            </repositories>
        </profile>
    </profiles>
    <activeProfiles>
        <activeProfile>flowable-artifacts</activeProfile>
    </activeProfiles>
</settings>
```

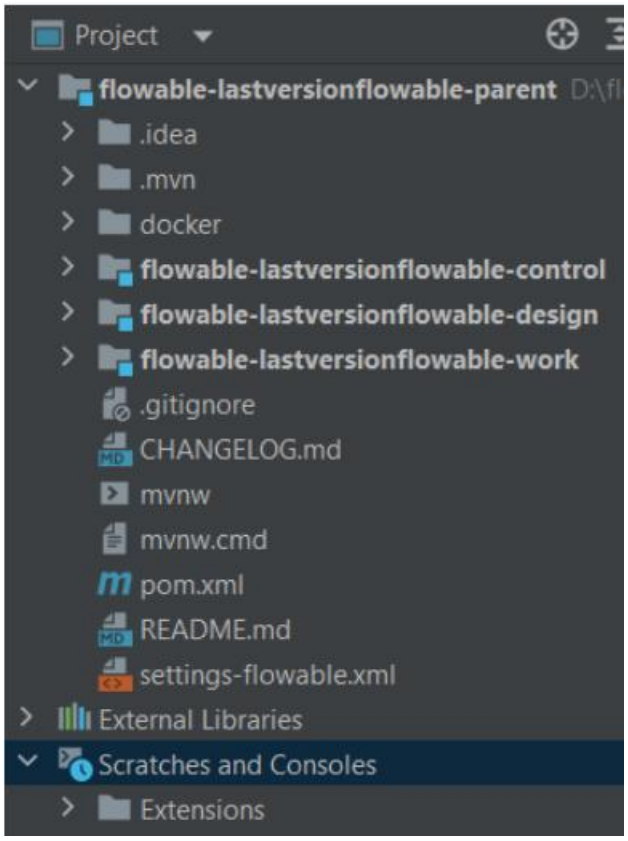
4. **Clone the Training Project:**
   - Clone the Flowable Getting Started repository from GitHub: `git clone https://github.com/flowable/flowable-training-getting-started`
5. **Open in Your IDE:**
   - Open the cloned project in your Java IDE (IntelliJ IDEA or similar).
   - Import the project using the IDE's import function.
6. **Start the Application:**
   - Your IDE should handle project import and configuration automatically.
   - Start any of the Spring Boot applications within the project.
7. **Accessing Flowable Applications:**
   - Once started, access Flowable applications locally:
     - Flowable Work: http://localhost:8090
     - Flowable Design: http://localhost:8091
     - Flowable Control: http://localhost:8092

This setup enables you to develop and explore Flowable locally.

Project Structure

The Flowable platform project structure adheres to the conventions of a typical multi-module Spring Boot application. At its core, the structure centers around three main applications: Control, Design, and Work, each residing in separate directories within the project.

The following overview should give you an idea of the most important files and folders in the demo project available **here**.



```
Project
flowable-lastversionflowable-parent D:\fl
  .idea
  .mvn
  docker
  flowable-lastversionflowable-control
  flowable-lastversionflowable-design
  flowable-lastversionflowable-work
  .gitignore
  CHANGELOG.md
  mvnw
  mvnw.cmd
  pom.xml
  README.md
  settings-flowable.xml
External Libraries
Scratches and Consoles
  Extensions
```

Root Directory:

- **pom.xml**: Maven project configuration file that manages dependencies and build settings. It facilitates the management of project dependencies and the updating of Flowable versions.
- **settings-flowable.xml**: Sample configuration file for Maven.

Applications Directory:

- **docker/**: Contains Docker Compose configuration (`docker-compose.yml`) defining services such as databases and Elasticsearch used by Flowable applications.
- **training-handson-***: Each Flowable application (Control, Design, and Work) is organized as a separate Maven module within its respective folder:
    - **Main Class**: `TrainingHandsOn[ApplicationName]Application.java` serves as the entry point for starting each application.
    - **Security Configurations**: Application-specific security configurations tailored to the needs of each module.
    - **Application Properties**: `application.properties` files along with environment-specific profiles (`application-postgres.properties`, etc.) are used to configure the applications.

The Spring Framework

Overview

The Spring Framework serves as the cornerstone of the Flowable Platform, offering a robust programming and configuration model tailored for Java-based enterprise applications. This chapter provides an overview of fundamental Spring concepts essential for effective development within the Flowable ecosystem.

Key Concepts and Features:
The Spring Framework facilitates the following core capabilities:

- **Dependency Injection (DI)**: Enables loose coupling by injecting dependencies into components, promoting modularity and testability.
- **Inversion of Control (IoC)**: Central to Spring's DI, IoC container manages object creation and lifecycle.
- **Aspect-Oriented Programming (AOP)**: Facilitates modularization of cross-cutting concerns, such as logging and transaction management.

- **Spring MVC and Spring WebFlux**: Provides web frameworks for developing robust, scalable web applications.

Additional Spring Features:
- **Data Access**: Includes transaction management, JDBC, ORM (Object-Relational Mapping), and XML marshalling.
- **Testing Support**: Mock objects, TestContext framework, Spring MVC Test, and WebTestClient for comprehensive testing.
- **Integration Capabilities**: Supports various integrations like remoting, JMS (Java Message Service), JCA (Java Connector Architecture), JMX (Java Management Extensions), email handling, task scheduling, caching, and observability.

While Spring MVC and Spring Data provide robust solutions, Flowable's integrated frontend (Flowable Work) and versatile data handling (Flowable Variables) reduce dependency on some of these modules.

However, understanding these capabilities allows developers to extend applications effectively, leveraging synergies between Spring and the Flowable Platform. This foundational knowledge equips you with essential tools for building scalable and efficient enterprise applications within the Flowable ecosystem.

Dependency Injection

This chapter introduces the core concepts of beans and dependency injection (DI) within the Spring Framework.

Beans and Dependency Injection
In Spring, a **bean** is simply a Java object managed by the Spring IoC (Inversion of Control) container. The IoC container is at the heart of Spring, responsible for

instantiating, configuring, and assembling beans. Instead of creating and managing objects manually, developers delegate this responsibility to the IoC container.

How Dependency Injection Works
Dependency Injection (DI) is a technique used by Spring to inject the dependencies of a bean automatically. This is typically achieved through constructor injection, where dependencies are passed into a bean's constructor when it's created.

Example Scenario
Imagine we have two service classes: `UserService` and `EmailService`. `UserService` relies on `EmailService` to send notifications. Without Spring, you might instantiate `EmailService` directly within `UserService`:

```
public class UserService {


    private EmailService emailService = new EmailService();


    public void notifyUser(String message) {
        emailService.sendEmail(message);
    }
}
```

In Spring, you define `EmailService` as a bean and inject it into `UserService` via constructor injection:

Magic of Dependency Injection
1. **Loose Coupling:** DI promotes loose coupling between components. `UserService` doesn't need to know how `EmailService` is created or configured; it only relies on its interface.
2. **Configuration Flexibility:** By defining beans and their dependencies in configuration classes (annotated with `@Configuration`) or using component scanning (`@ComponentScan`), Spring manages object creation and wiring.
3. **Testability:** In testing, you can inject mock implementations of dependencies for unit testing without modifying production code, enhancing testability and isolation.

Flowable and Spring Integration
Flowable leverages Spring's IoC container, allowing processes and cases to access Spring-managed beans directly, e.g. through Service Tasks. This integration enables Flowable workflows to directly interact with backend services through Spring's powerful dependency injection mechanism. Such services can then be used to perform any type of task you can think of.

Moreover, Flowable provides a range of out-of-the-box services for integration with the workflow engine, such as `runtimeService`, `repositoryService`, and others, simplifying interaction with the engine's core functionalities:

```
import org.flowable.engine.RuntimeService;
import org.springframework.stereotype.Service;


@Service
public class ProcessStarterService {


    private final RuntimeService runtimeService;


    // Constructor injection of RuntimeService, a Flowable service that allows you
to start/inspect processes
    public ProcessStarterService(RuntimeService runtimeService) {
        this.runtimeService = runtimeService;
    }


    public void startExampleProcess(String userId) {
        // Start a new process instance with a variable
        runtimeService.startProcessInstanceByKey("exampleProcess",
Map.of("userId", userId));
        System.out.println("Started process instance for user: " + userId);
    }
}
```

If you want to learn more about Beans and DI, you can check out the following, check out the following reference documentation:

- Dependency Injection
- IOC Container and Beans

Properties and Profiles

Spring Boot utilizes `application.properties` (or `application.yml`) files to manage configuration settings, allowing for flexible and environment-specific configurations. Flowable, as a Spring Boot application, uses this approach extensively for its own configuration needs.

Externalized Configuration in Spring
Spring Boot's configuration system enables developers to externalize configuration, making it easy to manage settings across different environments. Configuration properties are typically defined in `application.properties` files, which can include settings such as database connections, logging levels, and feature toggles.

Basic Example: `application.properties`

```
# Run the application on port 8081 using an h2 database
server.port=8081
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=flowable
spring.datasource.password=flowable
```

- **`server.port`**: Sets the port on which the application will run.
- **`spring.datasource.url`**: Configures the database connection URL.
- **`spring.datasource.username`**: Defines the database username.
- **`spring.datasource.password`**: Defines the database password.

Spring Profiles
Spring profiles allow you to define different configurations for different environments (e.g., development, testing, production). Each profile can have its own set of properties, enabling environment-specific settings.

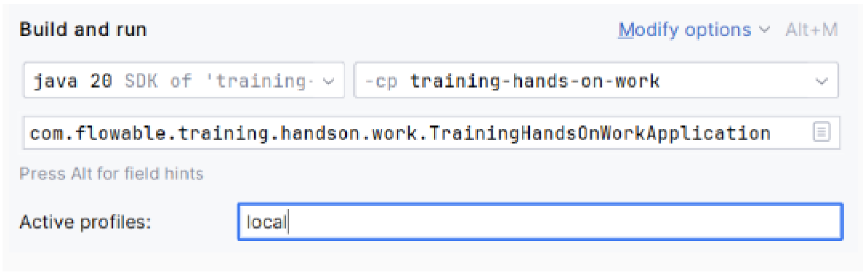
Using Profiles:

- **Default Profile**: The `application.properties` file is the default configuration.
- **Profile-Specific Files**: For environment-specific configurations, use `application-{profile}.properties`. For example, `application-local.properties` for local development and `application-prod.properties` for production.

Example: `application-local.properties`

```
# Development settings server.port=8081
spring.datasource.url=jdbc:h2:mem:devdb
```

You can activate profiles by setting the `spring.profiles.active` property or by setting them in your IDE:


Build and run

java 20 SDK of 'training-
-cp training-hands-on-work
com.flowable.training.handson.work.TrainingHandsOnWorkApplication

Active profiles: local

User Configurations
Users can also provide their own configuration settings in the `application.properties` file, which can be useful for customization and extending Flowable's capabilities. For example, users might add properties to control custom business logic or integrate additional services.

Example: Custom User Configuration

```
# Custom Configuration
my-app.default-currency=USD
my-app.notification.email.sender=noreply@example.com
```

These custom properties can then be accessed within your Spring application using `@Value` or through the `Environment` abstraction. You can also make use of Configuration Properties if you want your custom properties to be backed by a Java class.

Environment Property Resolution
Spring Boot resolves environment properties in a specific order of precedence:

1. **Command Line Arguments**: Properties passed as command-line arguments.
2. **Java System Properties**: Properties set with `-D` arguments.
3. **OS Environment Variables**: Properties set as environment variables.
4. **Profile-Specific Properties**: `application-{profile}.properties` files.
5. **Default Properties**: The default `application.properties` file.
6. **Application Defaults**: Hardcoded defaults within the application.

This hierarchy allows for flexible and overrideable configurations, making it easy to adapt to different deployment scenarios without changing the application code.

Flowable Development Basics

Service Tasks

In a previous module, you have learned about **Human** and **User Tasks**. They offer a handy way to set and present variables to a user. How do we handle scenarios where no user is involved?

This is where **Service Tasks** come in! Service Tasks in Flowable handle backend logic within processes and cases, executing operations without user involvement. They typically run in Flowable Work's Java backend and can be set up using two main approaches:

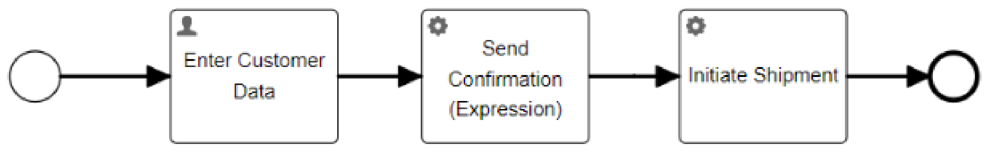
Types of Service Task Logic
1. **Expressions**
    - **Definition**: Use JUEL (Java Unified Expression Language) expressions to define the logic for a Service Task.
    - **Execution**: These expressions are evaluated directly in Flowable Work's Java environment and can interact with any Spring bean.
    - **Data Handling**: The result of the expression can be stored in a process or case variable for later use.
    - **Examples**:
        - `${firstName.toUpperCase()}`: Converts a string variable to uppercase.
        - `${myCrmService.updateCustomer(customerId, newCustomerInfo)}`: Calls the method *updateCustomer* on a Spring bean to update customer information.
2. **Delegate Expressions**
    - **Definition**: Use Java classes implementing `JavaDelegate` or `PlanItemJavaDelegate` interfaces to

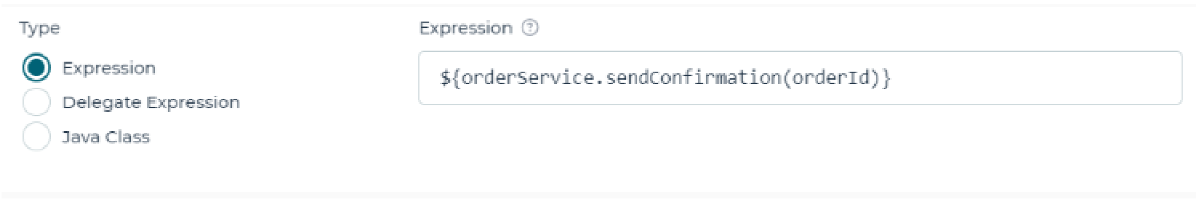
encapsulate task logic. By default, it is expected that such classes are exposed as singleton-scoped Spring beans.
    - **Execution**: These delegates are configured in Flowable Design to execute specific Java code when the task runs.
    - **Example**:
        - `${myCustomTask}`: Calls a custom Java class to perform specific logic when the task is executed. It will call the `execute(DelegateExecution execution)` method on the singleton instance of the class MyCustomTask. The task will have read and write access to the current process instance's execution, including all it's variables.

Example

In an order processing workflow, a Service Task might:

- **Using Expression**: `${orderService.sendConfirmation(orderId)}` calls a Spring bean method to send a confirmation email.
- **Using Delegate Expression**: A delegate class `ShippingDelegate` handles the logic to arrange shipment and update order status.

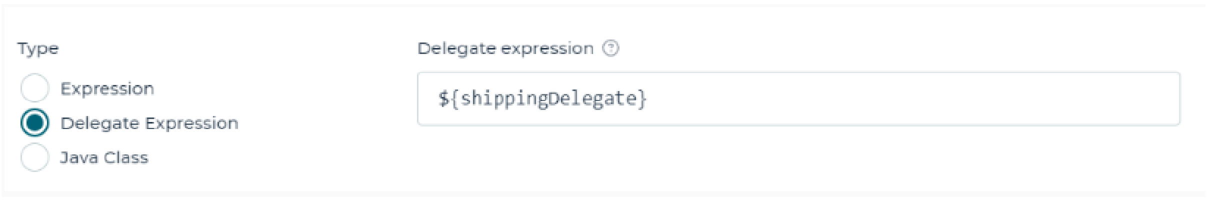



Type

[✓ Expression]
Delegate Expression
Java Class

Expression

`${orderService.sendConfirmation(orderId)}`


Type

Expression
[✓ Delegate Expression]
Java Class

Delegate expression

`${shippingDelegate}`

Note: For more controlled and reusable task logic, consider using the **Service Registry**, which allows defining a contract for inputs, outputs, and expressions, making service tasks easier to use for non-technical modelers. However, for simple or one-off operations, expressions and delegate expressions are typically sufficient.

Flowable Java API

Introduction to Flowable's Java API

When developing with Flowable, the primary method of interacting with process instances, tasks, and other workflow elements is through the **Java API** or **REST API**. Directly manipulating the database schema is strongly discouraged, as it could lead to inconsistencies in the internal state. Instead, the Java API provides a rich set of services for managing workflows, tasks, variables, and historic data in a way that ensures integrity and reliability.

Core Flowable Services
Flowable's Java API is based around Spring Services, which can be easily injected into your application. These services cover the primary interactions you'll have with Flowable's workflow, case, and decision engines. While there are specific services for each engine (e.g., CMMN for case management), the core concepts remain the same across engines.

Here are some of the most important services you'll use:

RuntimeService
The `RuntimeService` is one of the most commonly used services for interacting with running process instances. It allows you to:

- **Start** a new process instance
- **Complete** a process instance
- **Query** process instances
- **Manage variables** (set or delete variables tied to process instances)
- **Add or delete identity links** (associations between a user/group and a process instance)

Example - Starting a simple process instance:

```
runtimeService.startProcessInstanceByKey("myProcess");
```

Example - Starting a process instance with variables using the builder:

```
runtimeService.createProcessInstanceBuilder()
        .processDefinitionKey("myProcess")
        .variable("myVariable", "myValue")
        .variable("myVariable2", "myValue2")
        .start();
```

TaskService

The `TaskService` is responsible for managing user tasks within process instances. You can:

- **Start and complete** tasks
- **Query tasks** based on criteria like task name, assignee, or category
- **Set and delete task-specific variables**
- **Add or delete identity links** to tasks

Example - Completing all tasks in a specific category for the current user:

```
String currentUser = SecurityUtils.getCurrentUserSecurityScope().getUserId();
taskService.createTaskQuery()
        .taskCategory("myCategory")
        .list()
        .forEach(task -> taskService.complete(task.getId(), currentUser));
```

HistoryService

The `HistoryService` is used for querying past data related to processes and tasks. With it, you can retrieve historic information about completed or ongoing process instances, tasks, variables, and more.

Key functions include:

- Querying **historic process instances**
- Retrieving **historic tasks**
- Accessing **historic variables**

RepositoryService

The `RepositoryService` is mainly used for managing process definitions and deployments. You can:

- **Deploy new process definitions**
- **Search** for specific deployments or definitions
- **Delete** deployments or definitions

Example - Querying for a process instance:

```
runtimeService.createProcessInstanceQuery()
        .processDefinitionKey("myProcess")
        .variableValueEquals("myVariable", "myValue")
        .list();
```

Working with Flowable's CMMN and Other Engines

In addition to the BPMN engine, Flowable supports case management with its CMMN engine. The principles and services are consistent with the BPMN services, but tailored to cases rather than processes. For example:

- **CmmnRuntimeService**: Manages case instances

- **CmmnRepositoryService**: Manages case definitions and deployments

These services operate similarly to their BPMN counterparts, ensuring a consistent developer experience across engines.

What's more?
There are many additional aspects of Flowable that you can explore, such as other services, engine configuration, and advanced topics like listeners, but these are beyond the scope of this basic e-learning course. If you'd like to dive deeper, feel free to visit https://training.flowable.com to sign up for one of our developer-led training courses. You can also find detailed material in the documentation, such as learning more about Spring integration, configuration, and development concepts.

Flowable REST API

Flowable REST API Overview

Flowable provides a comprehensive REST API that allows developers to interact with all of Flowable's components, including the process, case, and decision engines. The API follows a resource-based design, typical of RESTful services, where each entity, like process instances or tasks, is represented by a unique endpoint.

The detailed API documentation can be accessed via Swagger for both the open-source and enterprise versions:

- Flowable OSS API Documentation
- Flowable Enterprise API Documentation

Resource-Based Structure
Flowable's REST API uses a structured path system that reflects the underlying services backing them. For example:

- To **start a new process instance**, you interact with the runtime engine, hence the endpoint is:
  `POST /process-api/runtime/process-instances`
- To **get information about a specific process definition**, which belongs to the repository (static metadata), the path is:
  `GET /process-api/repository/process-definitions/{id}`

This structure helps in understanding the logical flow of requests. Generally:

- **/runtime/** endpoints are used for running or active instances.
- **/repository/** endpoints are used for static information like process definitions or deployed resources.
  Depending on the engine you are interacting with, you may have to adapt the start of the path. For instance, to access a case definition, you have to call the following endpoint:

`GET /cmmn-api/cmmn-repository/case-definitions/{id}`

Common Operations and Examples

GET

The `GET` method is typically used to retrieve data, whether it's a collection of resources or a single item. For example:

- **Retrieve a specific process instance**:
  `GET /process-api/runtime/process-instances/{processInstanceId}`
  This returns details of a single process instance based on the provided `processInstanceId`.
- **List all tasks**:
  `GET /process-api/runtime/tasks`
  This retrieves a collection of tasks, possibly filtered by query parameters like `assignee` or `processInstanceId`.

POST

The `POST` method is used to create new resources or initiate actions. For example:

- **Start a new process instance**:
  `POST /process-api/runtime/process-instances`
  Body (JSON):
  `json { "processDefinitionKey": "myProcess", "variables": [ { "name": "myVariable", "value": "myValue", "type": "string" } ] }`
  This starts a new process instance using the provided process definition key and variables.

PUT

The `PUT` method allows updating existing resources or triggering actions on them. For example:

- **Update the category of a case instance**:
  `PUT /cmmn-api/runtime/case-instances/{caseInstanceId}`
  Body (JSON):
  `json { "category": "newCategory" }`
  This modifies the category of an existing case instance.

DELETE

The `DELETE` method is used to remove resources. For example:

- **Delete a deployment**:
  `DELETE /process-api/repository/deployments/{deploymentId}`
  This removes the specified deployment, including all associated resources like process definitions.

Paging and Sorting

Many of Flowable's API endpoints support pagination and sorting via the `start` and `size` parameters. This is especially useful when querying large datasets, such as listing tasks or process instances:

- **Example**:
  `GET /task-api/runtime/tasks?start=0&size=10&sort=name&order=asc`
  Retrieves the first 10 tasks, sorted by name in ascending order.

Caveats
Flowable's REST API offers a flexible and intuitive approach to interacting with the Flowable platform. However, their generic nature does not make them suitable for all situations. While all REST-APIs are gated behind an authorization layer, it may still be possible for an (authroized) malicious user to start a process instance through the regular REST API and inject arbitrary variables into the new process.

For that reason, an event-based approach or Custom REST Endpoints may be a better solution for some situations.

Custom REST Endpoints

In addition to the comprehensive REST API that Flowable provides, you might occasionally need to create custom REST endpoints to meet specific requirements. Here's a quick guide to creating `GET` and `POST` endpoints using Spring Boot's REST capabilities. These endpoints will demonstrate starting a new process instance and updating an existing process instance.

Setting Up REST Endpoints
Spring Boot makes it easy to create RESTful services. Here's how you can set up `GET` and `POST` endpoints in a Flowable application.

1. Create a REST Controller

First, create a new REST controller in your project.

```java
package com.example.flowable.rest;

import org.flowable.engine.RuntimeService;
import org.flowable.engine.runtime.ProcessInstance;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

@RestController
@RequestMapping("/api/process")
public class ProcessController {

    @Autowired
    private RuntimeService runtimeService;
    // POST endpoint to start a process
    @PostMapping("/start/{processKey}")
    public ProcessInstance startProcessInstance(
            @PathVariable String processKey,
            @RequestBody Map<String, Object> variables) {
        return runtimeService.startProcessInstanceByKey(processKey, variables);
    }

    // GET endpoint to get process instance details
    @GetMapping("/{processInstanceId}")
    public ProcessInstance getProcessInstance(@PathVariable String
processInstanceId) {
        return runtimeService.createProcessInstanceQuery()
                .processInstanceId(processInstanceId)
                .singleResult();
    }

    // PUT endpoint to update process variables
    @PutMapping("/{processInstanceId}")
    public void updateProcessInstanceVariables(
            @PathVariable String processInstanceId,
            @RequestBody Map<String, Object> variables) {
        runtimeService.setVariables(processInstanceId, variables);
    }
}
```

Explanation:

1. **Dependencies**: We use `RuntimeService` from Flowable to interact with process instances. Here, we are using an alternative approach to get the dependency in our class: `@Autowired`
2. **POST Endpoint**: `/start/{processKey}` starts a new process instance by key, passing variables in the request body.
3. **GET Endpoint**: `/{processInstanceId}` retrieves details about a specific process instance.
4. **PUT Endpoint**: `/{processInstanceId}` updates variables for an existing process instance.

2. Using the Endpoints

To use these endpoints:

- **Start a Process**: Send a `POST` request to `/api/process/start/myProcessKey` with a JSON body containing variables: `{ "applicantName": "John Doe", "loanAmount": 5000 }`
- **Get Process Details**: Send a `GET` request to `/api/process/{processInstanceId}` to retrieve process instance details.
- **Update Process Variables**: Send a `PUT` request to `/api/process/{processInstanceId}` with a JSON body containing the updated variables.

Note: Spring Boot automatically converts JSON request bodies to Java objects (and vice versa) using Jackson.

While Flowable provides a robust REST API out of the box, Spring Boot makes it straightforward to add custom endpoints when you need extra functionality. You can start new processes, retrieve process details, and update variables using simple REST methods. This flexibility allows you to tailor the Flowable engine's behavior to your specific application's needs.

Further Resources

Completion requirements

Additional Content
In addition to all the information we have seen during this course, we recommend the following links to extend your knowledge and get the best out of our Flowable developer course.

- Github page of Flowable: **https://github.com/flowable**
- Open Source (OSS) Documentation: **https://www.flowable.com/open-source/docs**
- Enterprise Documentation: **https://documentation.flowable.com/latest/**
- Tutorials:
    - BPMN Hello World: **https://documentation.flowable.com/latest/reactmodel/bpmn/introduction/hello-world**
    - CMMN Tutorial: **https://documentation.flowable.com/latest/reactmodel/cmmn/introduction/hello-world**
    - Backend Expressions: **https://documentation.flowable.com/latest/develop/be/be-expressions/index.html**
    - Java Extensions: **https://documentation.flowable.com/latest/develop/be/java-extensions**
    - Setting up a Flowable project from scratch:

HOW-TO // BEGINNER // DEV
Setting Up a Development Project
Flowable
with Valentin Zickner
Solution Architect
at Flowable
WATCH NOW

Play Video