Platform, Services, and Utilities

Generated on: 2021-10-21 17:20:00 GMT+0000

SAP Commerce | 1905

PUBLIC

Original content: https://help.sap.com/viewer/d0224eca81e249cb821f2cdf45a82ace/1905/en-US

Warning

Writing a Hello World CronJob

Here you will create a simple cron job named **HelloWorldCronJob**, which uses a job that prints out log information.

Cron jobs have some advantages over using services, because they may be called not only on demand, but also at some defined time and may be run asynchronously. You may store the logged output for every call of a cron job. Moreover, logic of a cron job can be aborted on demand.

Creating a New Extension

To be able to create and use cron jobs, first install and initialize the Platform. The **cronjob** functionality is then automatically available as a part of the Platform.

To make sure that **HelloWorldCronJob** is kept apart from other parts of SAP Commerce, use a new extension to implement it. See also Installation Based on Specified Extensions for information on how to do it. Use the following settings for the new extension:

- extension.name=cronjobtutorial
- extension.package=de.hybris.cronjobtutorial

The new `cronjobtutorial` extension is generated in *<HYBRIS_BIN_DIR>* `/custom` directory. Do not forget to reference the new extension in the `localextension.xml` file. See also Installation Based on Specified Extensions for information on how to do it.

Declaring HelloWorldCronJob

Declare a custom `CronJob` type called `HelloWorldCronJob`. Do this in the `cronjobtutorial-items.xml` file located in `resources` folder in your new extension. Add there a subtype of `CronJob` type like in the following example:

`cronjobtutorial-items.xml`

```
<items  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xsi:noNamespaceSchemaLocation="items.xsd">

        <itemtypes>
                <itemtype generate="true"
                   code="HelloWorldCronJob"
                   jaloclass="de.hybris.cronjobtutorial.jalo.HelloWorldCronJob"
                   extends="CronJob"
                   autocreate="true">
                </itemtype>
        </itemtypes>
</items>
```

The subtype of general `CronJob` type can provide some additional attributes, which are used only by this subtype. For more details, read items.xml. If no additional attributes are needed, you may skip the definition.

Rebuild SAP Commerce by calling `ant` in the *<HYBRIS_BIN_DIR>* `\platform` directory. `HelloWorldCronJobModel` is generated in the `bootstrap/gensrc` directory. For more information see Models, **Model Class Generation** section.

Create JobPerformable

To define the logic for a job, in the `de.hybris.cronjobtutorial` package create a new class extending `AbstractJobPerformable` and override the `perform` method. In this tutorial, the job should create log information. You may also implement the `JobPerformable` interface directly. The `AbstractJobPerformable` class is used to define the job

as performable and non-abortable by default and to provide access to some services. It implements the `JobPerformable` interface.

`MyJobPerformable.java`

```
public class MyJobPerformable extends AbstractJobPerformable<HelloWorldCronJobModel>
{
	private static final Logger LOG = Logger.getLogger(MyJobPerformable.class.getName());

	@Override
	public PerformResult perform(final HelloWorldCronJobModel cronJobModel)
	{
		LOG.info("*********************************");
		LOG.info("Greeting from MyJobPerformable!!!");
		LOG.info("*********************************");

		return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
	}
}
```

Rebuild the SAP Commerce by calling the `ant` in the *<HYBRIS_BIN_DIR>* `\platform` directory. The new `MyJobPerformable` has to be defined as a Spring bean in the `cronjobtutorial-spring.xml` file located in the `resources` folder of the `cronjobtutorial` extension:

`cronjobtutorial-spring.xml`

```
<bean id="myJobPerformable" class="de.hybris.cronjobtutorial.MyJobPerformable"
	parent="abstractJobPerformable"/>
```

After changing the Spring configuration, restart the hybris Platform and perform a system update. For details see Initializing and Updating SAP Commerce. A system update is needed, because during the phase of essential data creation, for each Spring definition of a class implementing the `JobPerformable` interface, a `ServicelayerJob` instance gets created and the `code` attribute of the job is set to the name of the Spring bean. Hence, using an instance of `ServicelayerJobModel` with the `code` attribute set to the `myJobPerformable` job will call the new implemented `myJobPerformable`.

Creating Trigger and Running HelloWorldCronJob

To run the new cron job, you need to perform the following steps:

1. Get the job instance of `ServicelayerJobModel` having `springid` attribute set to `myJobPerformable`.
2. Create and configure an instance of `HelloWorldCronJob`.
3. Assign the job to the cron job and set attributes for the cron job.

See an example of a test class that you can create in your `cronjobtutorial` extension:

`HelloWorldIntegrationTest.java`

```
public class HelloWorldIntegrationTest extends ServicelayerTest
{

	private static final Logger LOG = Logger.getLogger(HelloWorldIntegrationTest.class.getName(

	@Resource
	CronJobService cronJobService;

	@Resource
	ModelService modelService;

	@Resource
	FlexibleSearchService flexibleSearchService;
```

```
List<ServicelayerJobModel> servicelayerJobModelList = Collections.EMPTY_LIST;
ServicelayerJobModel servicelayerJobModel = null;
HelloWorldCronJobModel helloWorldCJ = null;

@Before
public void setUp()
{
        //The update of the JUnit tenant creates automatically an instance of the defined M
        //Search for it
        ServicelayerJobModel sjm = new ServicelayerJobModel();
        sjm.setSpringId("myJobPerformable");
        try
        {
           servicelayerJobModel = flexibleSearchService.getModelByExample(sjm); //searching
        }
        catch(ModelNotFoundException e)
        {
           //The cronjob functionality in the processing extension creates for each JobPerf
           //You just create a job here
           servicelayerJobModel = modelService.create(ServicelayerJobModel.class);
           servicelayerJobModel.setSpringId("myJobPerformable");
           servicelayerJobModel.setCode("myJobPerformable");
           modelService.save(servicelayerJobModel);
           //Keep in mind that creating models in the catch clause is bad style
        }

        // Create a CronJob and set the servicelayerJob
        helloWorldCJ = modelService.create(HelloWorldCronJobModel.class);
        helloWorldCJ.setActive(Boolean.TRUE);
        helloWorldCJ.setJob(servicelayerJobModel);
        modelService.save(helloWorldCJ);

        //Below is just to show how to create a trigger but not really necessary for this J
        createTrigger(helloWorldCJ);
}

@Test
public void testIfThePerformableExist()
{
        //Check if there is an instance of myJobPerformable
        assertNotNull("***************No performable with springID *myJobPerformable* found
                        + "Update your JunitTenant to let create an instance!", servicelayerJ
}

@Test
public void testExecuteThePerformable()
{
        //Check if setup works correctly
        assertNotNull("***************The in set upcreated CronJob is null?", helloWorldCJ)

        //Perform the CronJob once for the test
        cronJobService.performCronJob(helloWorldCJ);

        //Wait for the result to be written
        try
        {
                Thread.sleep(2000);
        }
        catch (final InterruptedException e)
        {
                e.printStackTrace();
        }

        LOG.info("*************** lets wait 2 seconds for the result  ***************");

        //Test if the job was executed successfully, if it fails here then try to extend th
        assertEquals("*************** The perfromable has not finished successfull or more
                        CronJobResult.SUCCESS, helloWorldCJ.getResult());

}

//Create a trigger just to show how to implement it
public void createTrigger(final HelloWorldCronJobModel helloWorldCJ)
```

```
        {
                final TriggerModel triggerModel = modelService.create(TriggerModel.class);
                triggerModel.setActive(Boolean.TRUE);
                triggerModel.setMinute(new Integer(1));
                triggerModel.setCronJob(helloWorldCJ);
                modelService.save(triggerModel);
        }

}
```

The assigned `JobPerformable` type should match the `CronJob` type. It is important to use an appropriate `springid`, so that the `ServicelayerJobModel` points to an existing `JobPerformable` bean definition.

Triggers can be created using cron expressions. When you create a trigger, you should then assign it to the cron job. The code sample above presents how to create a trigger that executes the cron job once a minute. Below is a sample with CronExpression:

```
//Choose the cron expression for starting the job every 10 seconds
        triggerModel.setCronExpression("0/10 * * * * ?");
```

Running the `HelloWorldCronJobModel` results in the following output on a console:

```
*********************************
Greeting from MyJobPerformable!!!
*********************************
```

Writing an Abortable Job

Because job execution may take a lot of time, you may wish to abort a job. To do this, add a code snippet for checking if you wish to abort a job. Because you can not easily and safely abort a `JobPerformable` at any time unless you kill the thread, it is necessary to place hooks for the abort before stages that you expect to be time-consuming.

The tutorial uses the job created in the HelloWorldCronJob Tutorial that is adequately adjusted as described below.

Adjusting MyJobPerformable

The execution of `MyJobPerformable` from HelloWorldCronJob Tutorial does not take a lot of time. Let us modify the job to extend its execution time by adding a `for` loop:

`MyJobPerformable.java`

```
public class MyJobPerformable extends AbstractJobPerformable<HelloWorldCronJobModel>
{
  private L10NService l10nService;

  @Required
  public void setL10nService(final L10NService l10nService)
  {
    this.l10nService = l10nService;
  }

  @Override
  public PerformResult perform(final HelloWorldCronJobModel cronJob)
  {

    for (int i = 0; i <= 1000; i++)
    {
      try
      {
        System.out.println("Greeting '" + l10nService.getLocalizedString(cronJob.getMessage())
                        + "' from MyJobPerformable for " + i + " times.");
```

```
        Thread.sleep(5000);
      }
      catch (final InterruptedException e)
      {
        Thread.currentThread().interrupt();
      }
    }
    //the following will be executed when the loop is finished
    return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
  }

}
```

The job displays the message in a console:

```
Greeting 'Hallo' from MyJobPerformable for 1 times.
Greeting 'Hallo' from MyJobPerformable for 2 times.
Greeting 'Hallo' from MyJobPerformable for 3 times.
Greeting 'Hallo' from MyJobPerformable for 4 times.
Greeting 'Hallo' from MyJobPerformable for 5 times.
        ...
```

After 5000 seconds the job succeeds and the following message is displayed in the console:

```
Greeting 'Hallo' from MyJobPerformable for 5000 times.
```

Overriding isAbortable Method

By default each job is not abortable. To enable the abort feature, you should override the `isAbortable` method in `MyJobPerformable`:

`MyJobPerformable.java`

```
@Override
public boolean isAbortable()
{
  return true;
}
```

Alternatively, if you extend the `AbstractJobPerformable`, you may override a bean property value in Spring configuration:

`cronjobtutorial-spring.xml`

```
<bean id="myJobPerformable" class="de.hybris.cronjobtutorial.MyJobPerformable"
parent="abstractJobPerformable" >
  <property name="l10nService" ref="l10nService" />
  <property name="abortable" value="true"/>
</bean>
```

Checking Abort Request

Currently the job is marked as abortable, which allows to activate the abort flag for the cron job, similarly to the `interrupt` flag for `Thread`. When implementing an abortable job it is important to regularly check if the user has requested to abort the job. If true, the abort flag is set, the job should be stopped, and the cron job should have a proper status and result. You should decide when to check the flag and what to do if the flag is set, for example, to clean up. Time of processing between verifications should be moderate.

The following code sample calls the `clearAbortRequestedIfNeeded` of the `AbstractJobPerformable`. It checks if the given `cronJob` is requested to be aborted and has `REQUESTABORT` flag set to `true`. If so, the flag is set to `false` again and

the job execution is prematurely stopped.

`MyJobPerformable.java`

```
if(clearAbortRequestedIfNeeded(cronJob))
{
   //abort the job

   //do some clean-up

  return new PerformResult(CronJobResult.ERROR, CronJobStatus.ABORTED);
}
```

After including the `if` condition in `perform` method, the `MyJobPerformable` looks as follows:

`MyJobPerformable.java`

```
public class MyJobPerformable extends AbstractJobPerformable<HelloWorldCronJobModel>
{
        private L10NService l10nService;

        @Required
        public void setL10nService(final L10NService l10nService)
        {
                this.l10nService = l10nService;
        }

        @Override
        public PerformResult perform(final HelloWorldCronJobModel cronJob)
        {

        for (int i = 0; i <= 1000; i++)
        {
                try
                {
                        System.out.println("Greeting '" + l10nService.getLocalizedString(cronJob.ge
                                        + "' from MyJobPerformable  for " + i + " times.");
                        Thread.sleep(5000);

                        if (clearAbortRequestedIfNeeded(cronJob))
                        {
                                System.out.println("The job is aborted.");
                                return new PerformResult(CronJobResult.ERROR, CronJobStatus.ABORTED
                        }
                }
                catch (final InterruptedException e)
                {
                        Thread.currentThread().interrupt();
                }
         }

        return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
        }

        @Override
        public boolean isAbortable()
        {
                return true;
        }

}
```

Aborting the Job

To abort the job, you should call the method:

```
cronJobService.requestAbortCronJob(cronJob);
```

The cron job results then with **ERROR**, its status is set to **ABORTED** and the following message is displayed in the console:

```
The job is aborted.
```

Importing CronJob Instances Using ImpEx

Using ImpEx for creating instances of `CronJob` is useful if you want to define it declaratively, and not by creating code that must be compiled and deployed. This is worth applying especially for a system set up in combination with ImpEx Import for Essential and Project Data, as the data setup is often adjusted.

The tutorial presumes that you have already created `MyJobPerformable` and configured it as a Spring bean, as described in the HelloWorldCronJob Tutorial.

Update the system to automatically create the instance of `ServicelayerJob` related to defined `myJobPerformable` bean from Spring context.

Importing CronJobModel Instance

Create a cron job by importing an instance of `HelloWorldCronJob` type and reference it to the `JobPerformable` instance:

```
INSERT_UPDATE HelloWorldCronJob;code[unique=true];job(code);message;sessionLanguage(isocode);
;myHelloWorldServicelayerCronJob;myJobPerformable;greeting.key;en;
```

This example creates an instance of `HelloWorldCronJobModel` with an assigned `ServicelayerJob` instance, and some custom localized message key.

Creating Trigger

If you wish to start your cron job automatically, you need to define a trigger:

```
INSERT_UPDATE Trigger; cronJob(code)[unique=true]; cronExpression
; myHelloWorldServicelayerCronJob; 0 30 10-11 ? * WED,FRI
```

This ImpEx script creates an instance of `Trigger` assigned to `myHelloWorldServicelayerCronJob` and a cron expression that causes the trigger to start the cron job at 10:30, 11:30, every Wednesday and Friday.