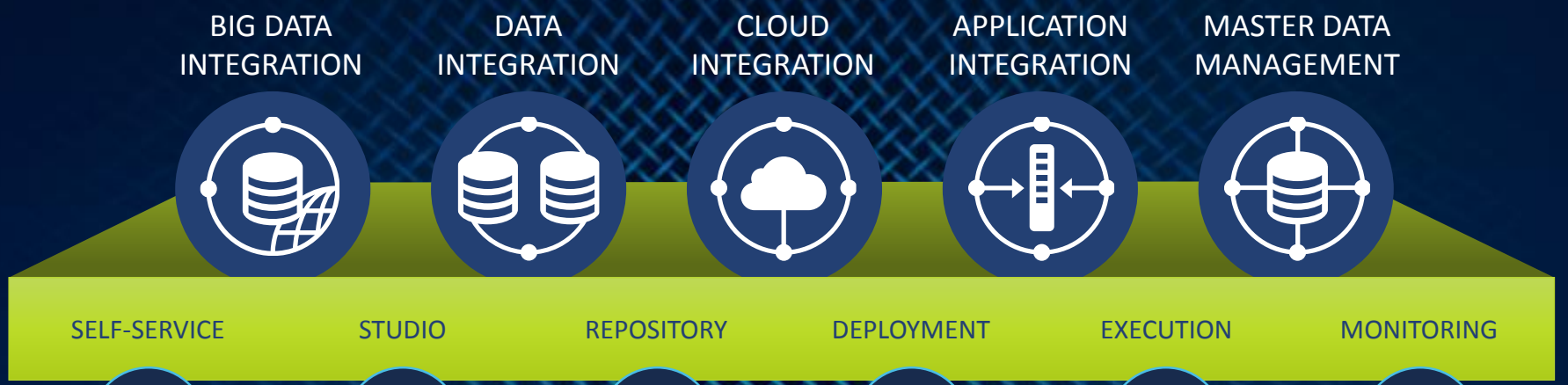
Data Integration Basics

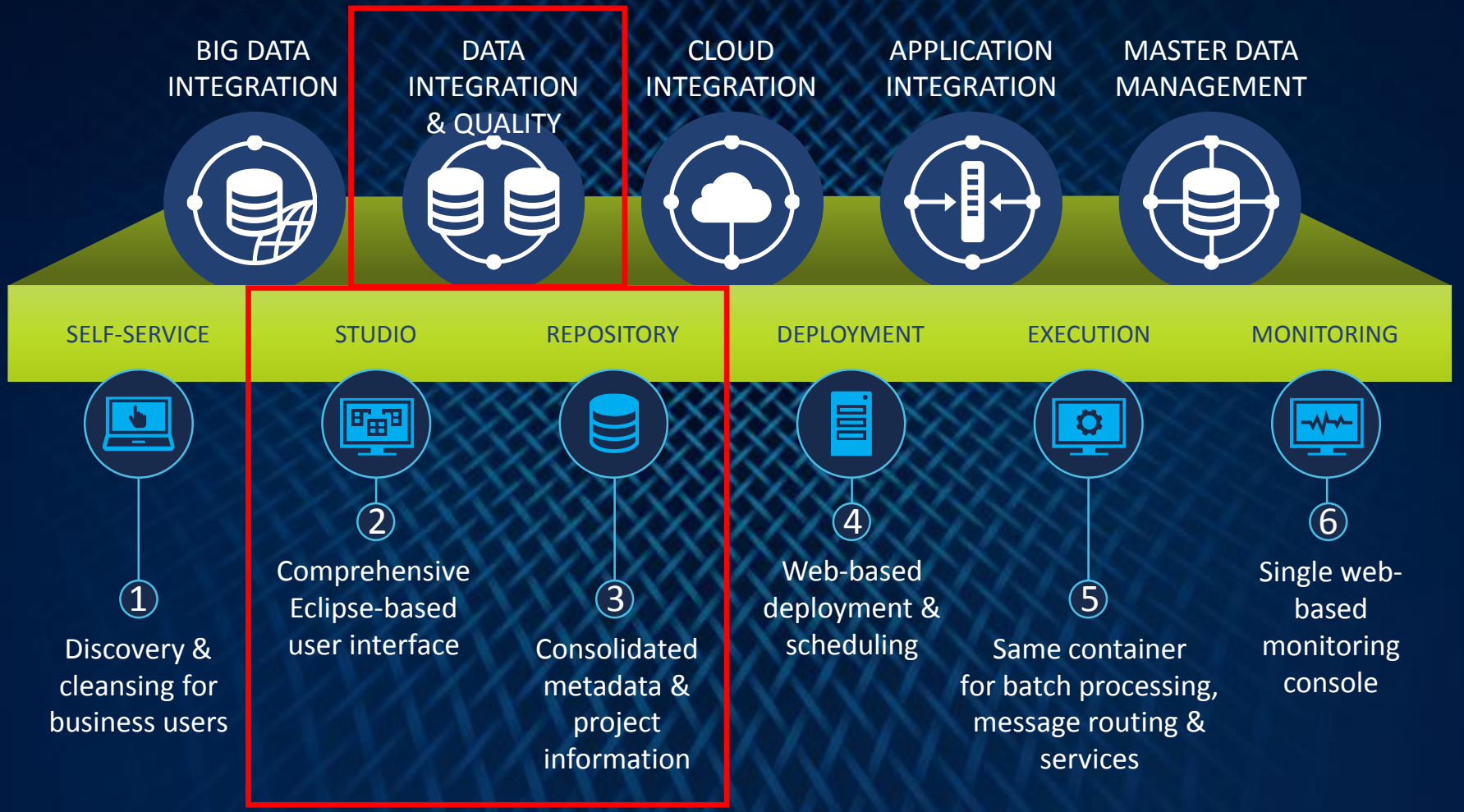
Version 6.2

talend | Data Fabric



1. Discovery & cleansing for business users
2. Comprehensive Eclipse-based user interface
3. Consolidated metadata & project information
4. Web-based deployment & scheduling
5. Same container for batch processing, message routing & services
6. Single web-based monitoring console

talend | In this course, we focus on



Course Table of Contents


talend

Introduction to Talend Data Integration

Version 6.2

Objectives

- Define Data Integration
- Explore typical project use cases
- Describe key benefits of Data Integration

Why Data Integration?

More data every day

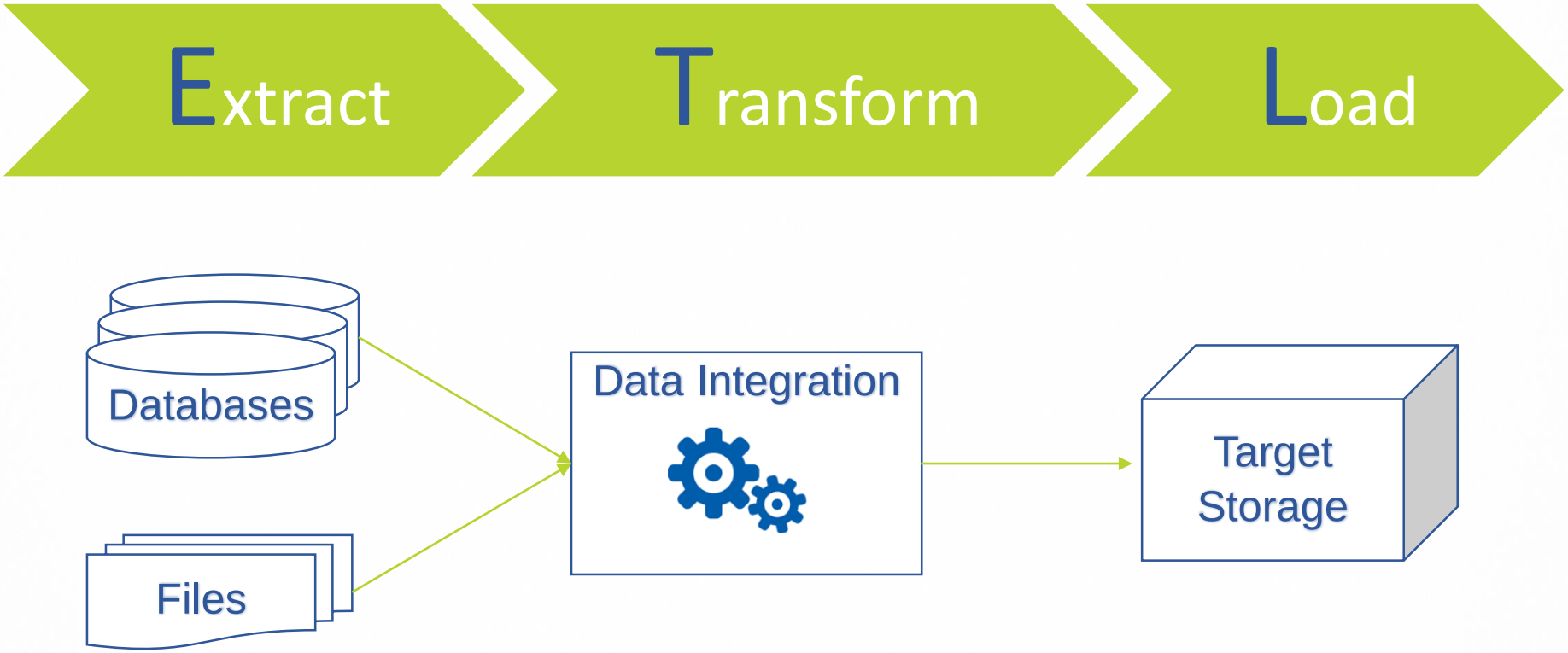
- Different formats
- Different database structures and models
- Different standards
- Different systems

Adds value to data

- Removes mistakes and corrects data
- Adjusts data from multiple sources to be used together
- Structures and aggregates data to be used by BI tools

What is Data Integration?

- Allows data coming from different systems to be merged
- Implementation is simplified by using an ETL tool

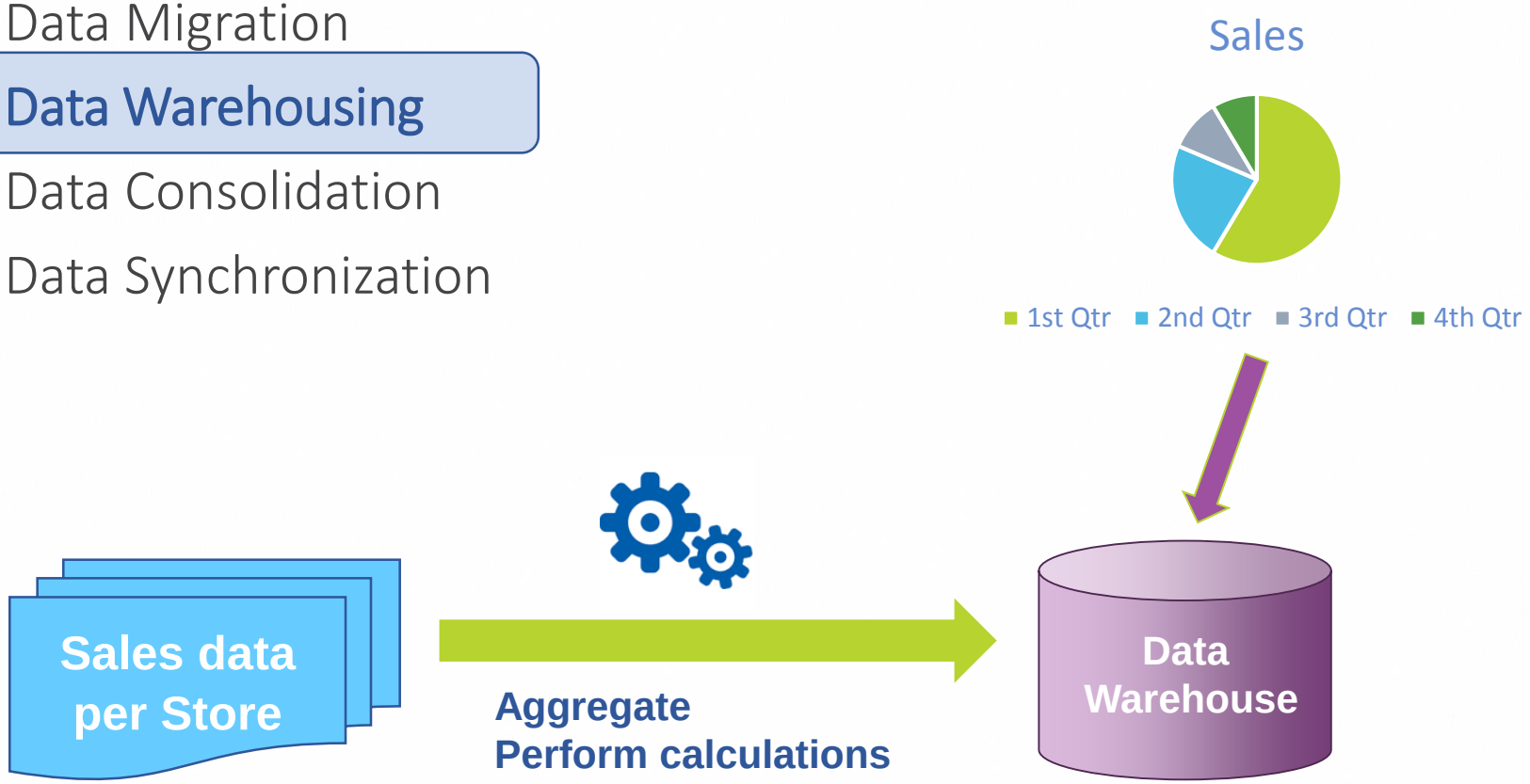


Typical Use Cases

- Data Migration
- Data Warehousing
- Data Consolidation
- Data Synchronization

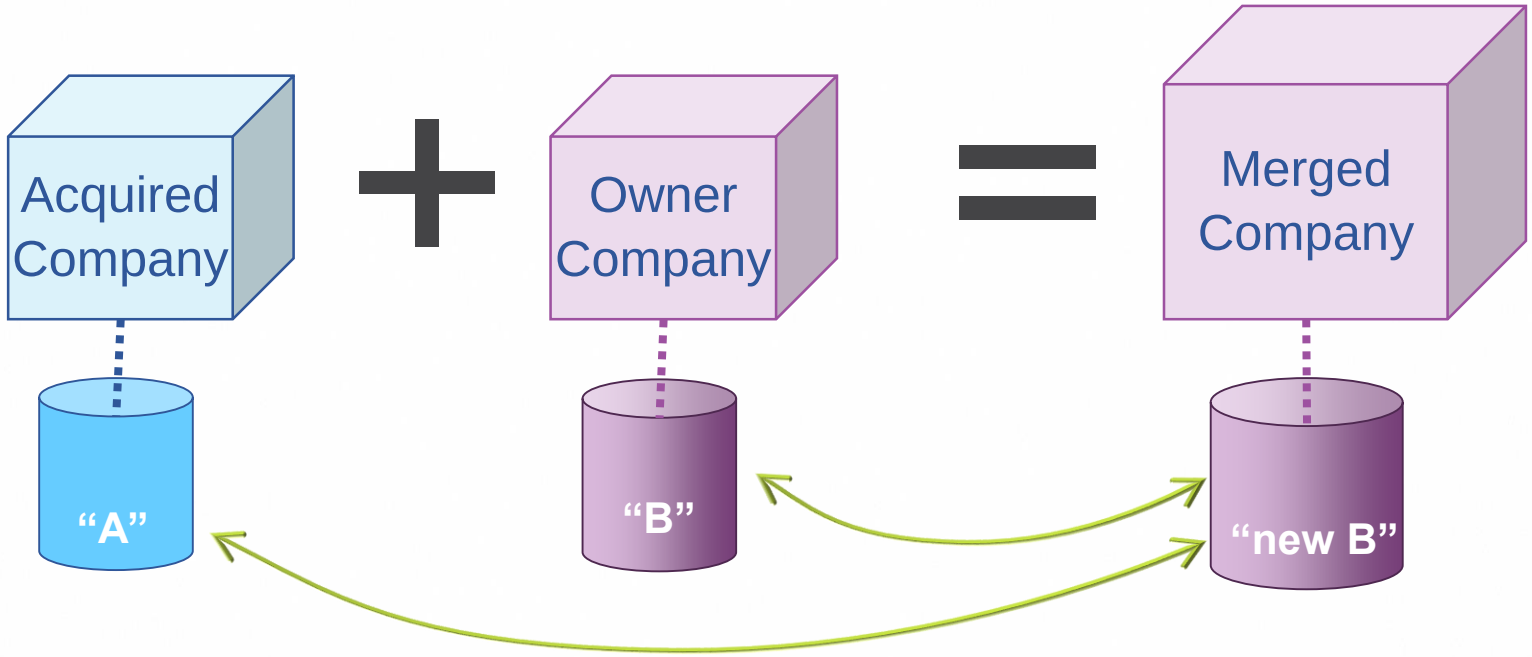
Typical Use Cases

- Data Migration
- **Data Warehousing**
- Data Consolidation
- Data Synchronization



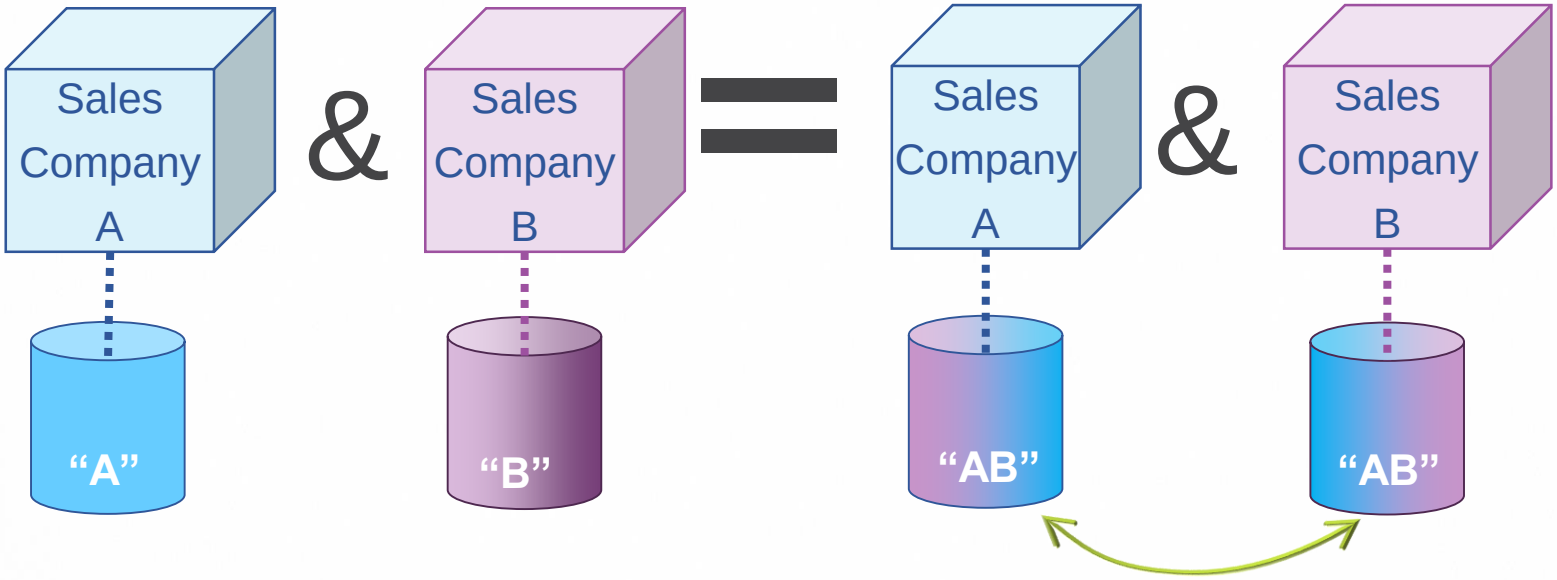
Typical Use Cases

- Data Migration
- Data Warehousing
- **Data Consolidation**
- Data Synchronization



Typical Use Cases

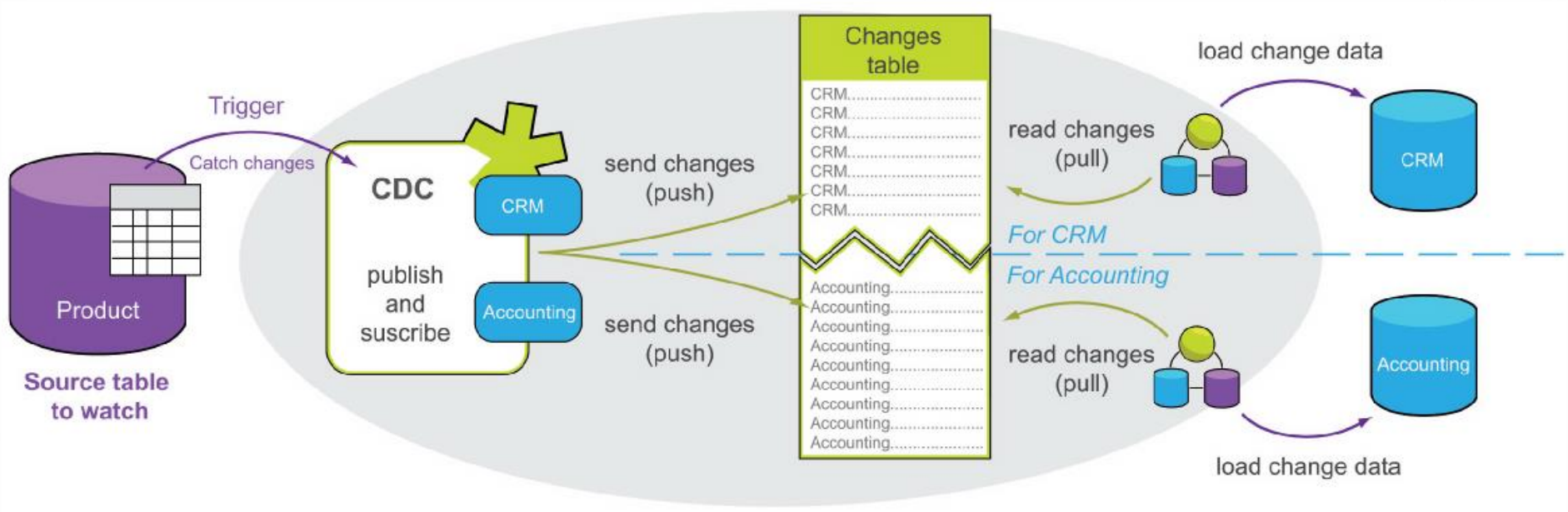
- Data Migration
- Data Warehousing
- Data Consolidation
- **Data Synchronization**



Integration Using Delta Only

Change Data Capture Scenario

- The **Change Data Capture** is based on the DB Journal or triggers
- A **Changes table** is used to track changes
- **Publish / Subscribe** mechanism



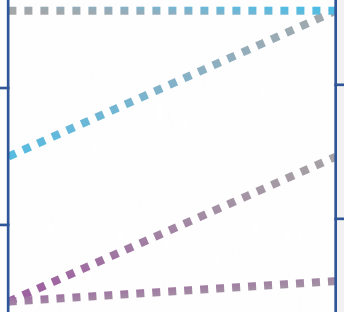
Key benefits of Data Integration

- Connectivity to source and target data systems
- Scalability and performance
- Transformation flexibility
- Data Cleansing components
- Logging and Exception Handling
- Easy integration with Web Services

Data Integration Power

- Splitting or merging fields
  - Over 800 components to assist you

First Name	John
Last Name	Smith
Credit Card	VISA 12 3456 7891



Full Name	John Smith
Card Type	VISA
Card Number	12 3456 7891

Data Integration Power

- Data Standardization
  - Dates
  - Units of measurement

Source systems

Product	Length	Date
Cable 1	5 m	25/01/2015

Product	Length	Date
Cable 2	38.2 Ft	2015-16-02

Target system

Product	Length	Date
Cable 1	5 m	25 Jan 2015
Cable 2	10 m	16 Feb 2015

Data Integration Power

- Join Data
  - Complete the data with information from external sources

Source system

Product	Quantity	Zip Code	Customer
P167987	13	10004	C987674
P167987	16	10005	C987856

Reference file / WS

Zip Code, City
10005, New York
10004, New York
….

Target system

Product	Quantity	Zip Code	City	Customer
P167987	13	10004	New York	C987674
P167987	16	10005	New York	C987856

Data Integration Power

- Aggregate data
  - Perform several calculations
  - Group and order elements as needed

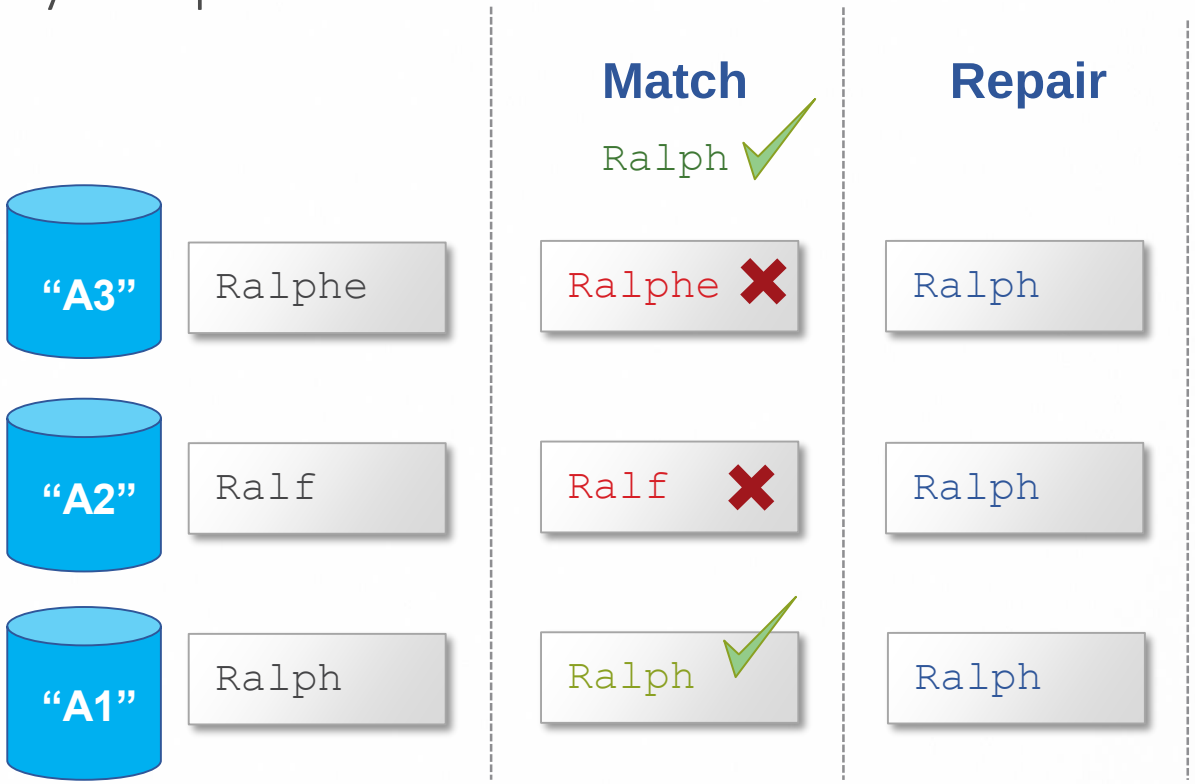
Product	Quantity	Zip Code	City	Customer
P167987	13	10004	New York	C987674
P167987	16	10005	New York	C987856

Aggregated data to be analyzed
(grouped by city and by product)

Product	Quantity	City
P167987	29	New York

Data Integration Power

- Supports Data Deduplication in Data Quality and MDM projects
  - Several Data Quality components available



Why Talend Data Integration?

Respond Faster to Business Needs

Design Faster

Use Talend Studio to design data integration jobs with a drag and drop user interface

Collaborate Better

Enable teams of developers to collaborate better using the Team Repository

Cleanse Earlier

Use profiling and data matching to understand and cleanse data

Manage More

Centralize deployment and manage projects through a web-based Administration Console

Scale Easier

Scale up with a high availability environment

Getting Started

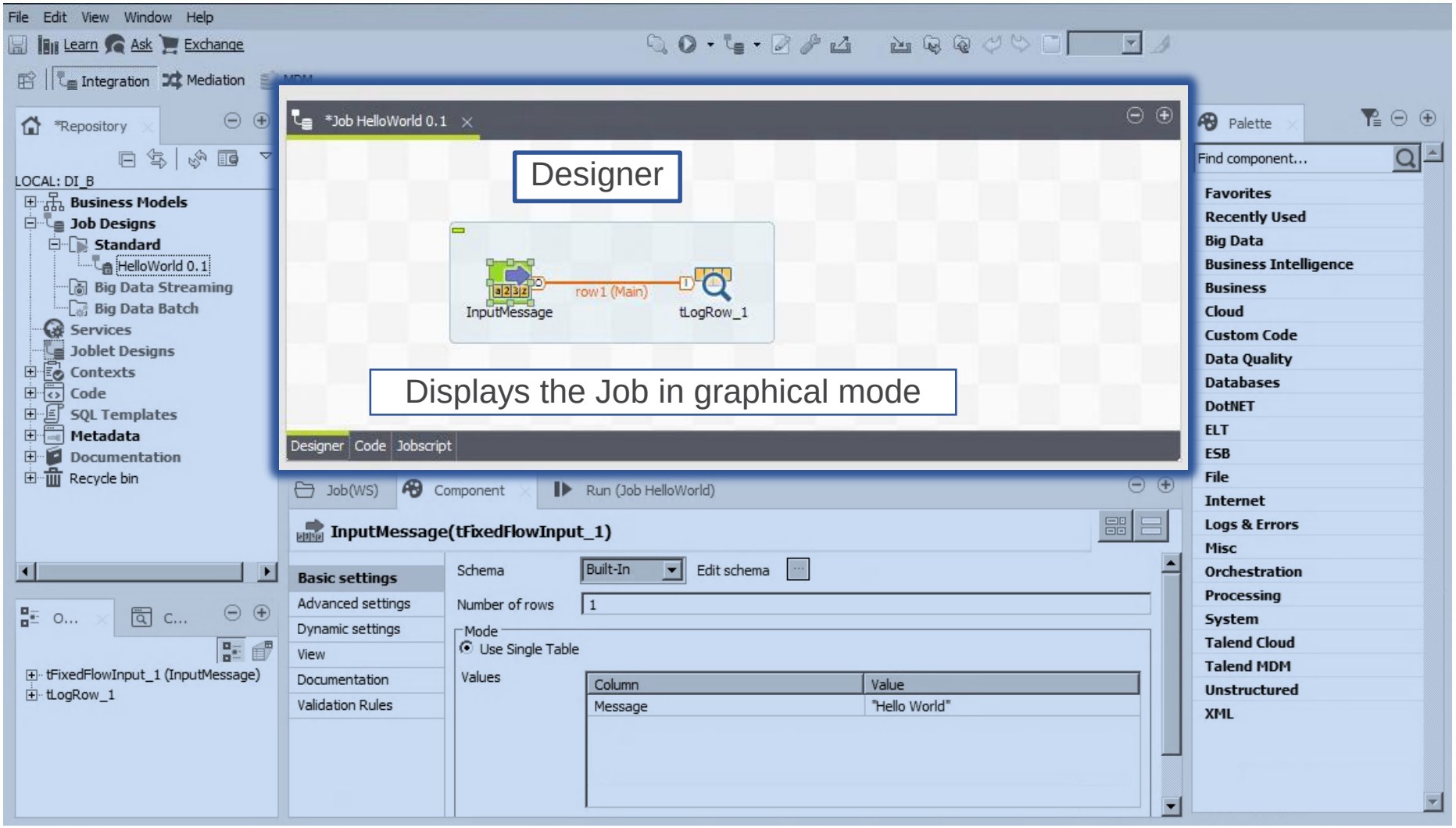
Talend Studio for Data Integration

Objectives

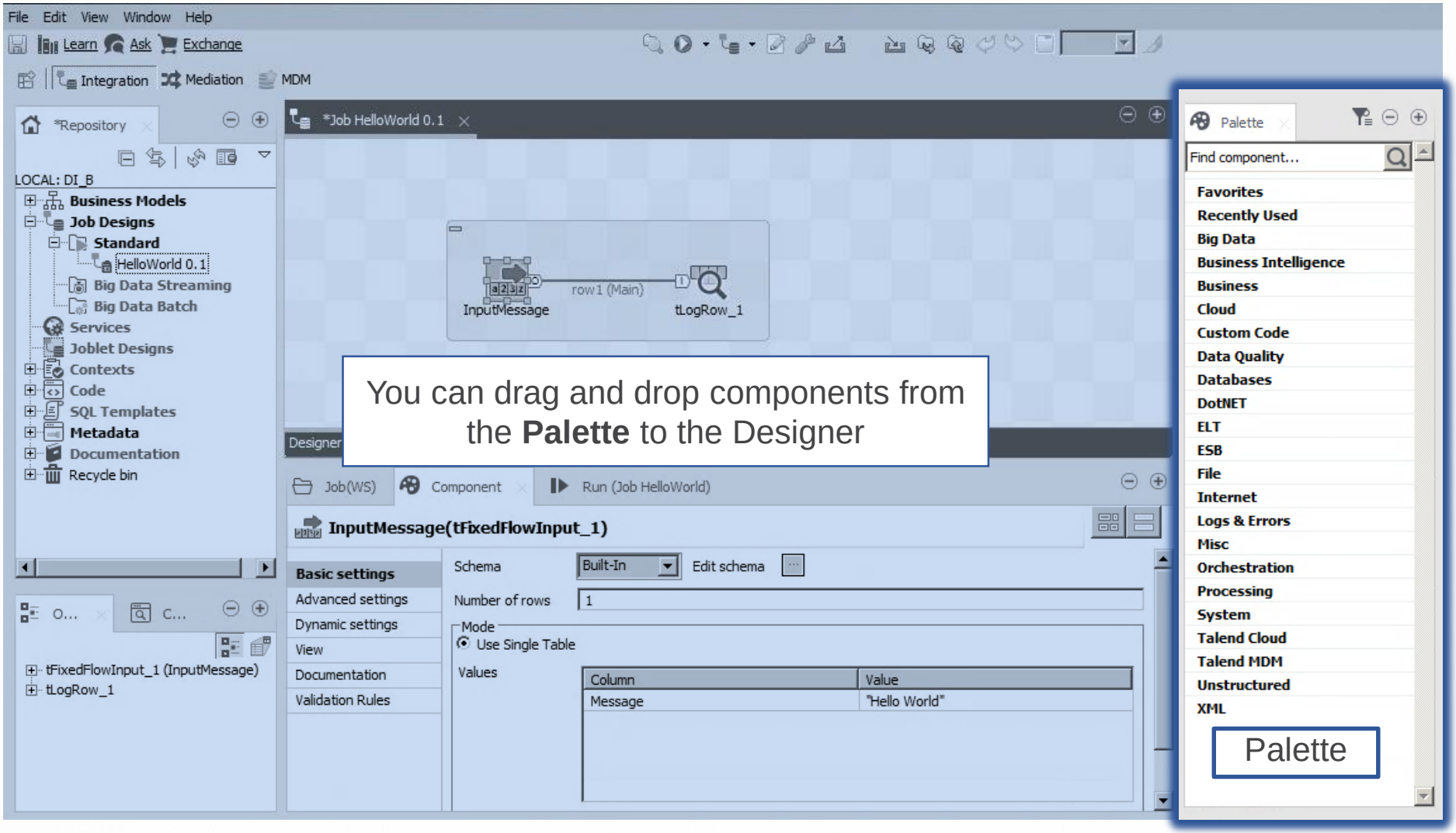
By the end of this lesson, you will be able to:

- Start the Talend Studio
- Create a new Job
- Find and add components
- Connect and configure components
- Get help with components
- Run a Job

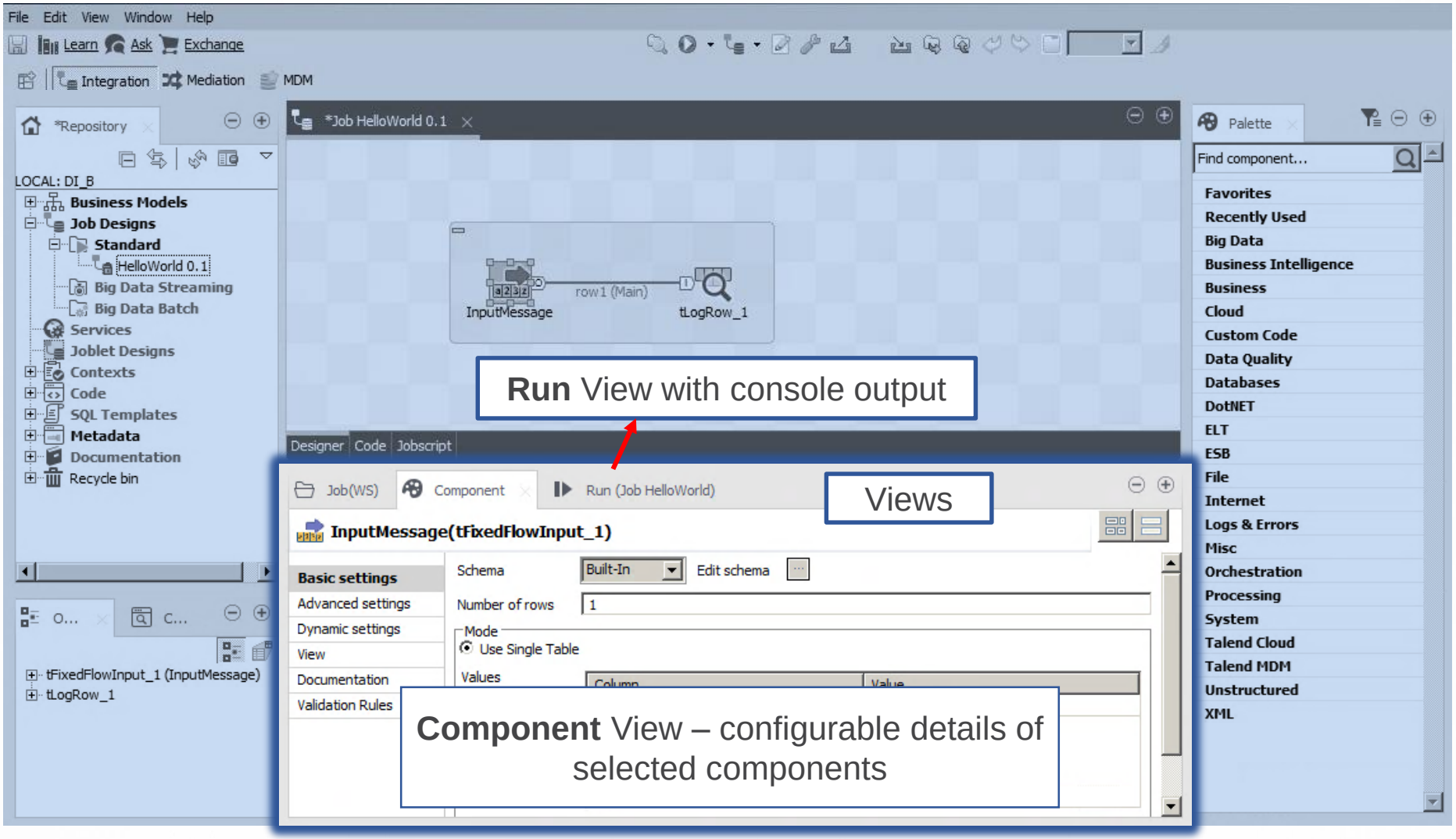
Talend Studio Main Window



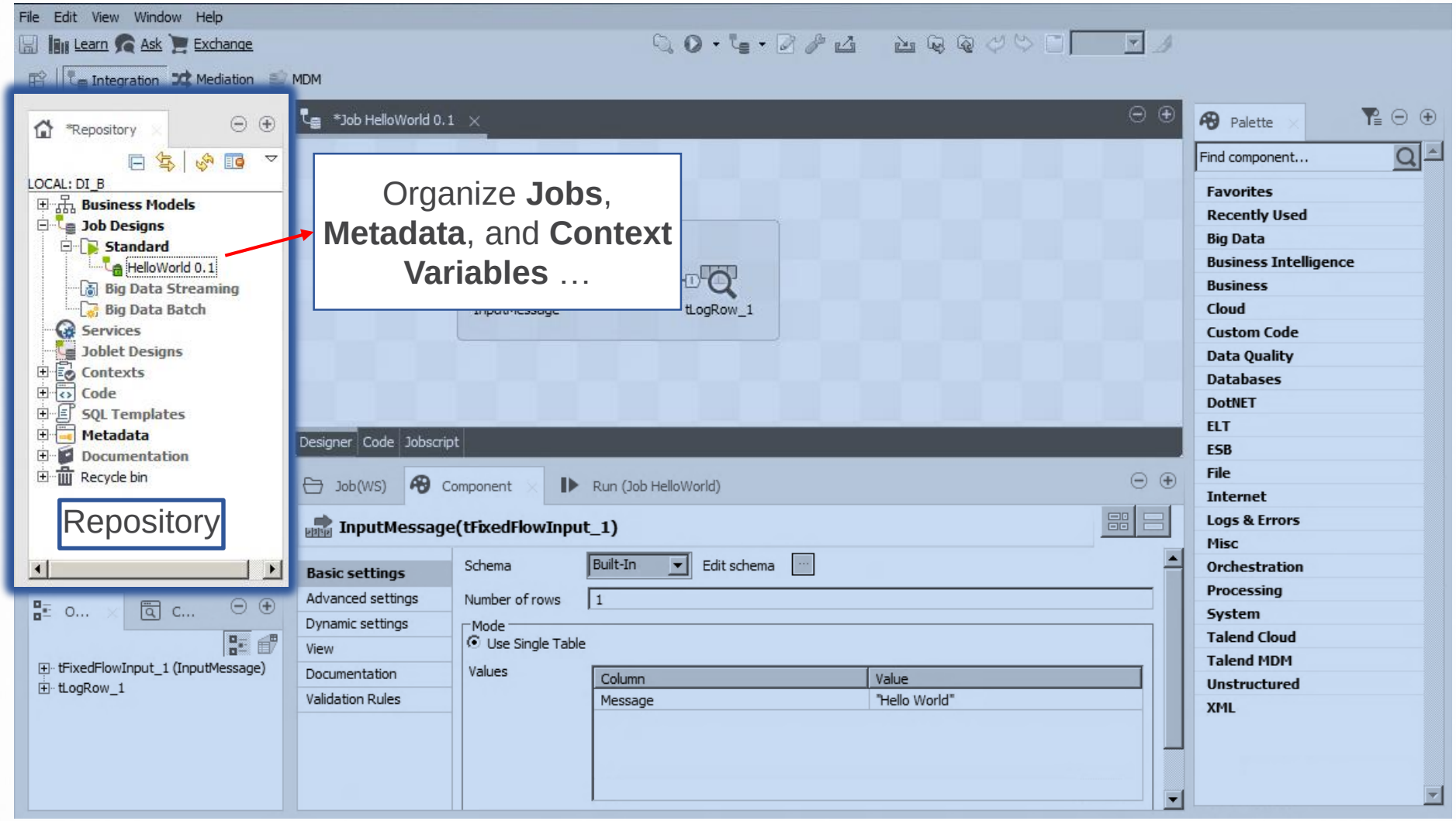
Talend Studio Main Window



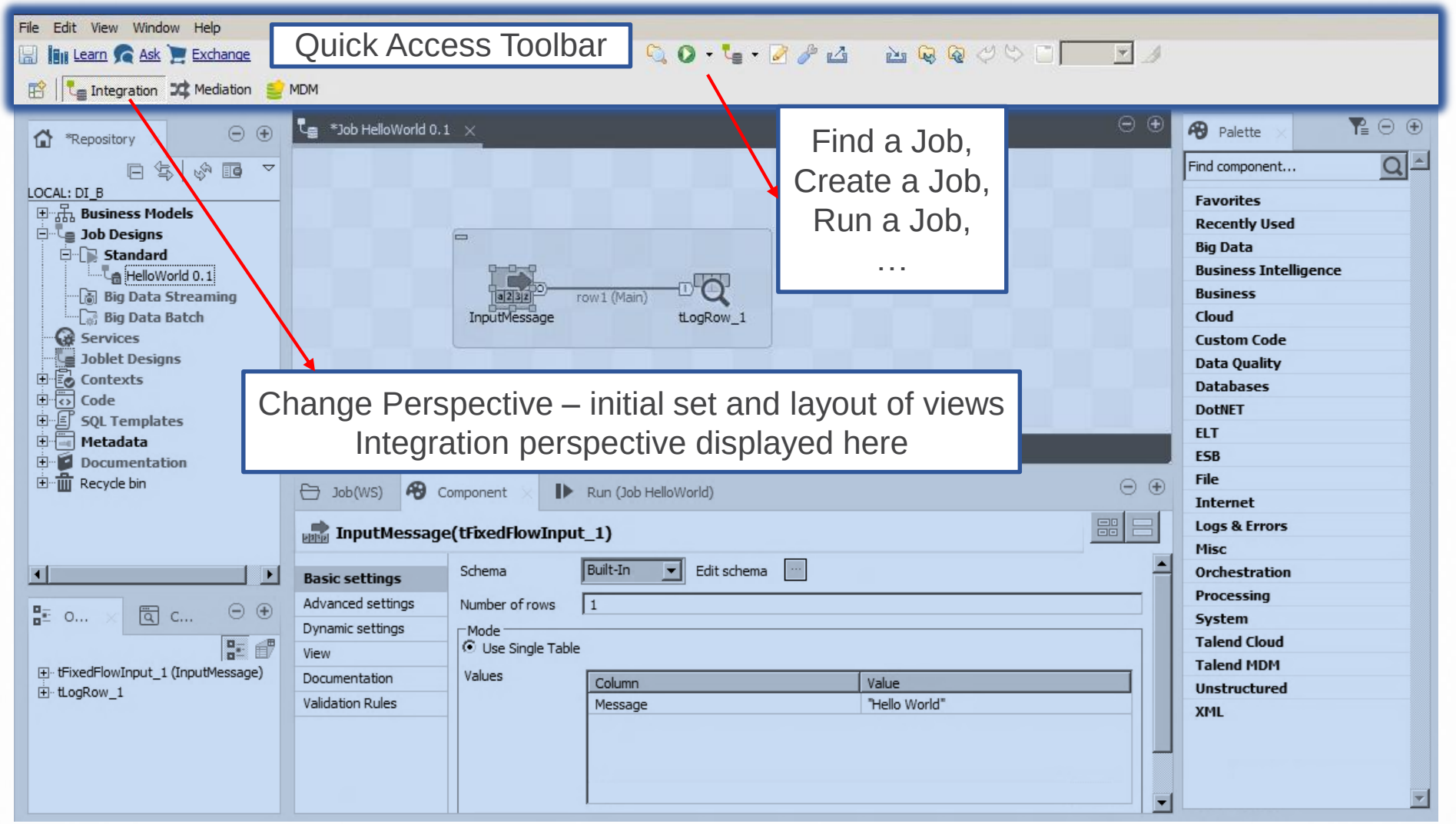
Talend Studio Main Window



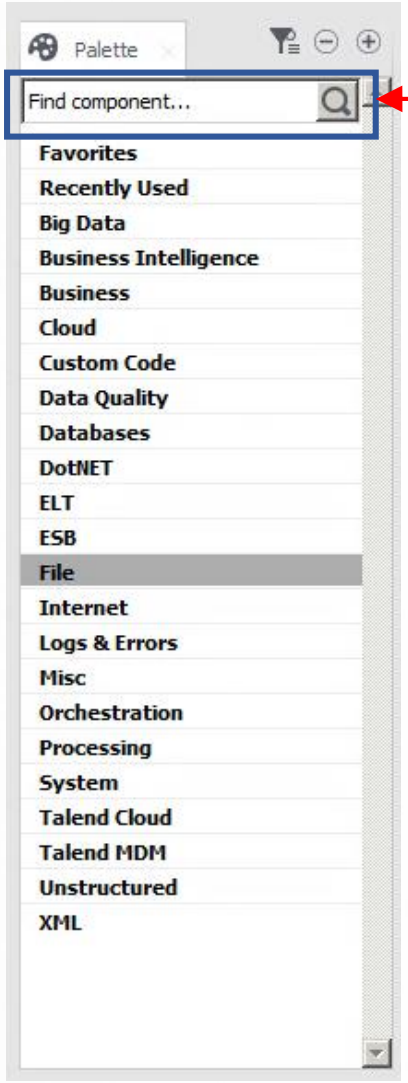
Talend Studio Main Window



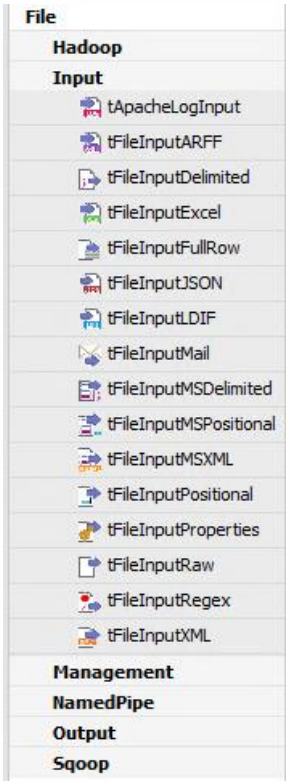
Talend Studio Main Window



User Interface - Components



Components can be found in the **Palette** using their names or keywords.



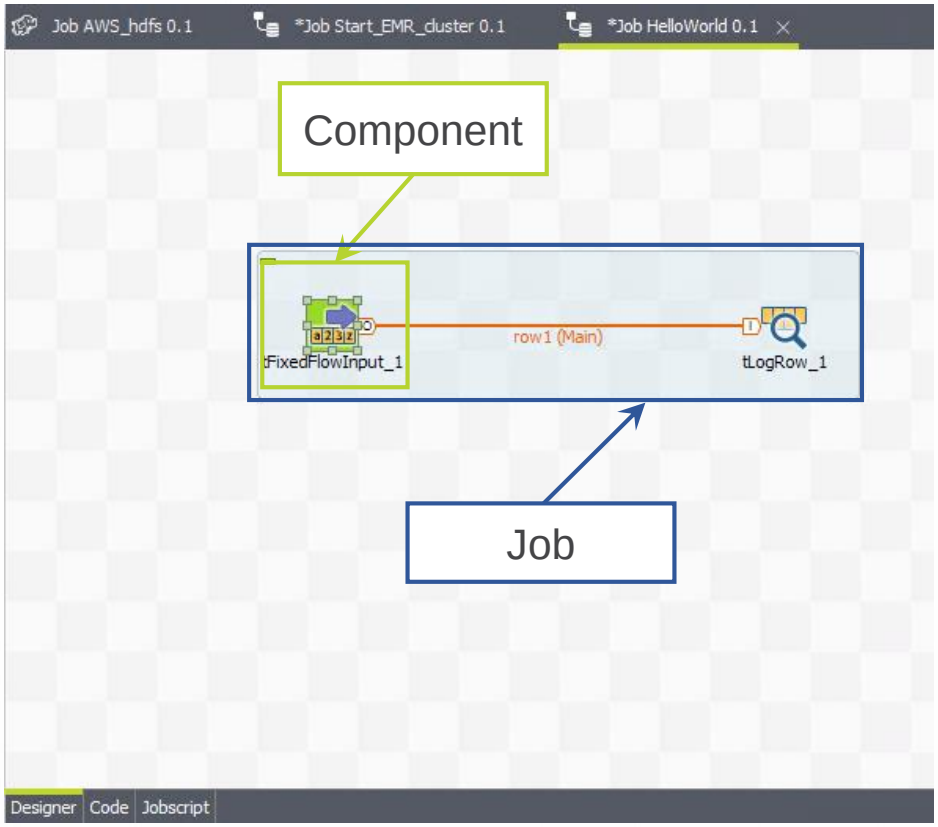
Components are organized by **family**:

- File
- Databases
- Processing
- Logs & Errors
- Orchestration
- …

Types inside of families:

- Input
- Output
- …

What Is a Talend Job?

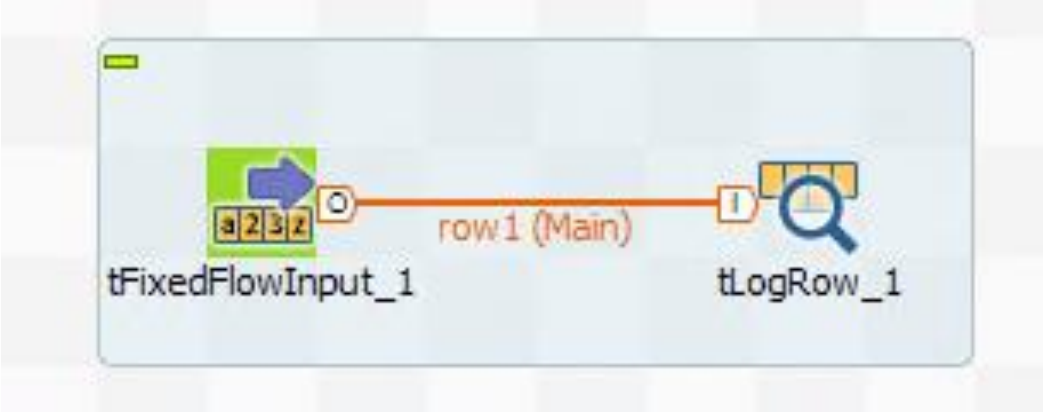


- **Component** – Executable part of a Job. Talend provides 900+ pre-built components to accomplish specific tasks.
- **Job** – One or more components connected together (graphically) that generate java code

Build a Job

Example *Hello World* Job

Display a Hello World message



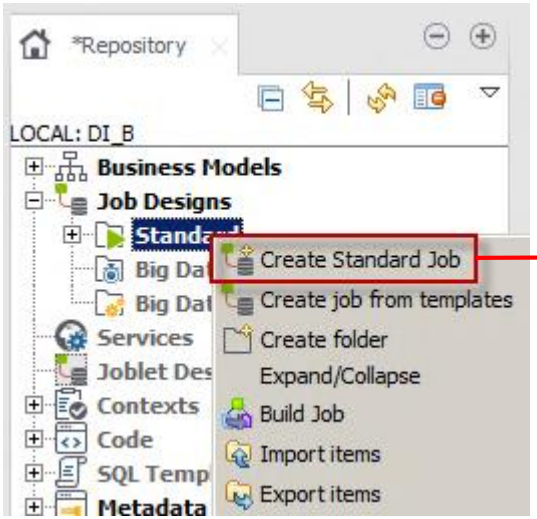
- **tFixedFlowInput**
  Used to define a message for an internal variable

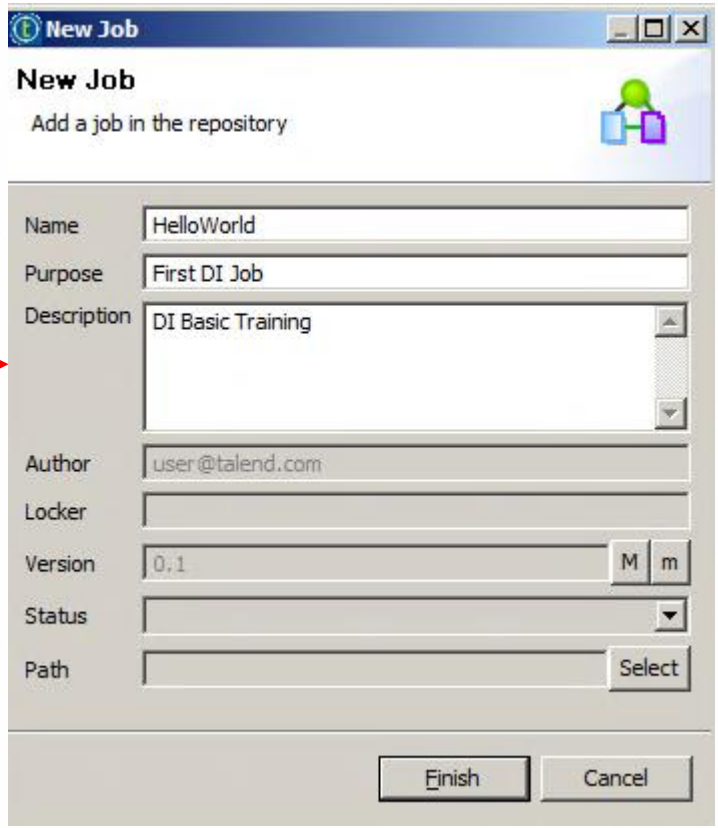
- **tLogRow**
  Used to display the message in the execution console

Build a Job

Create a Job

- Repository View > Job Designs > Right-click Standard
- Select **Create Standard Job**





- **Name**: no spaces or special characters
- Fill out both **Purpose** and **Description**

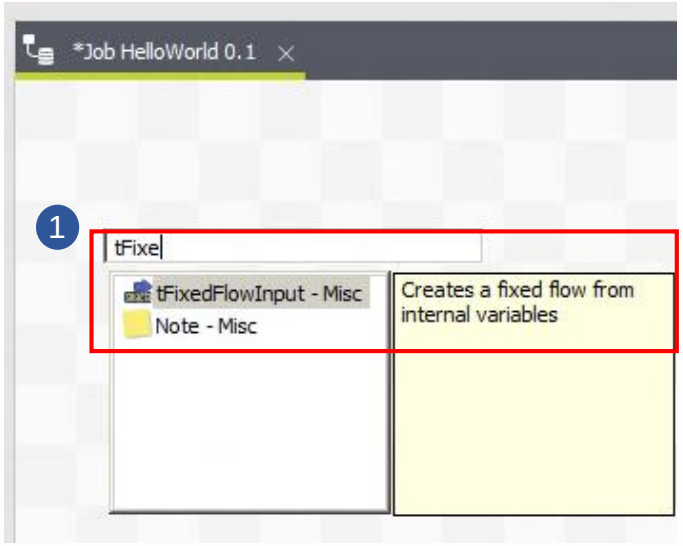
Build a Job

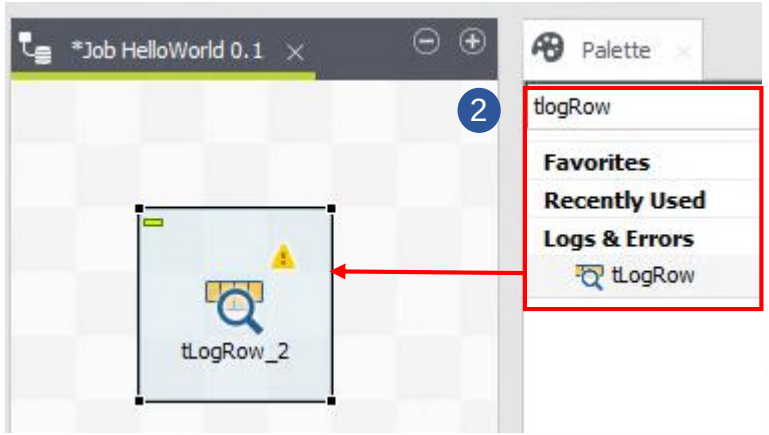
Add components

Two options:

1. Type the first characters of the component in the design space
   - Auto-complete function provides a list of choices (select the component)

2. Search for the component in the Palette
   - Drag and drop it to the Designer





Build a Job

Connect components

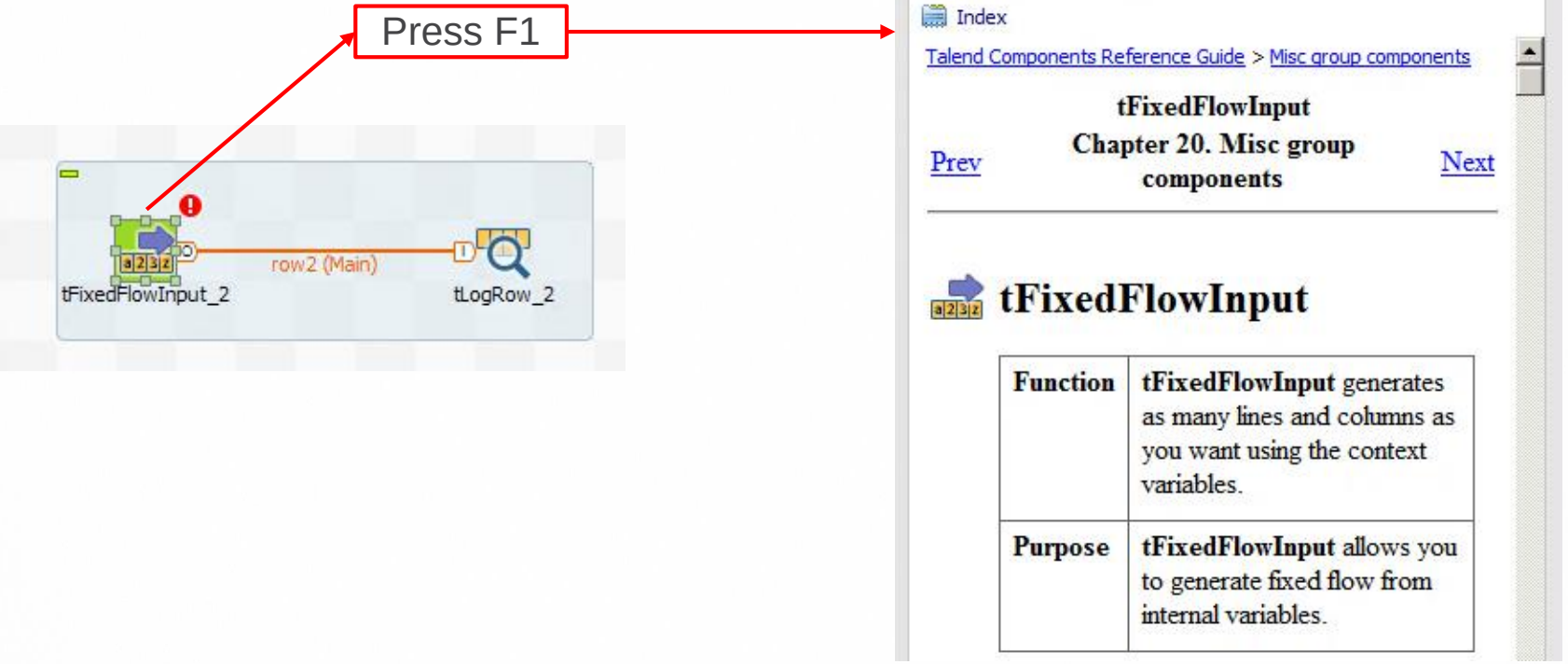
- To create a direct link between two components:
  - Right click the component
  - Select Row > Main
  - Click on the second component

- Main rows can also be created by **holding a right click** from the first component to the second one

Build a Job

Online context sensitive help

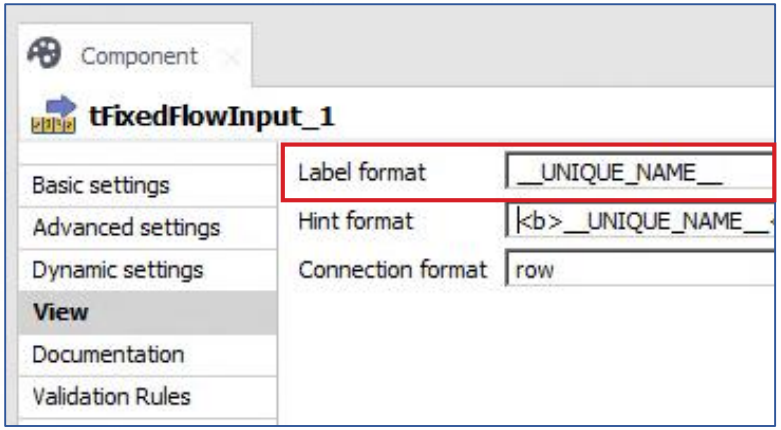
- To open a component's Help page:
  - Select component
  - Press F1

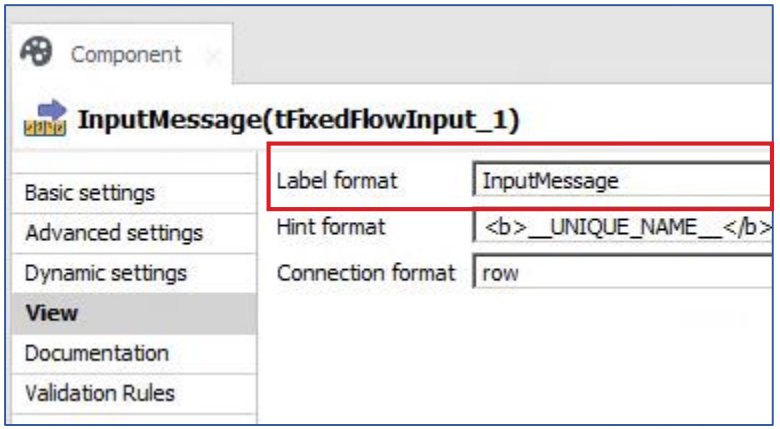


Build a Job

Rename a component

- Use the **Component** View
  - **View** tab
  - Change the component's name in the **Label format** box

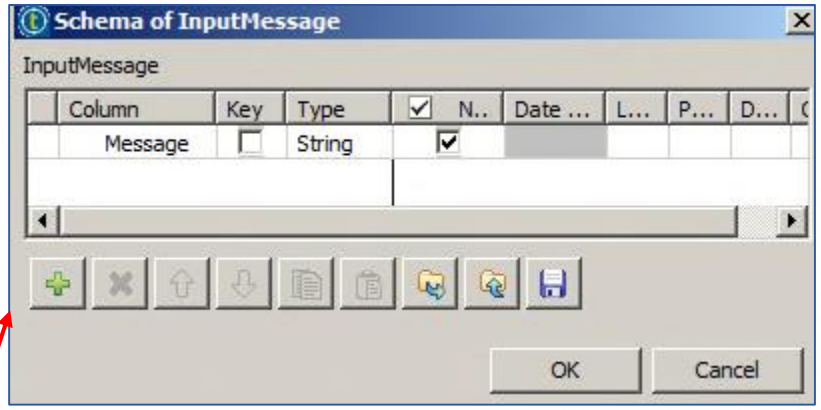


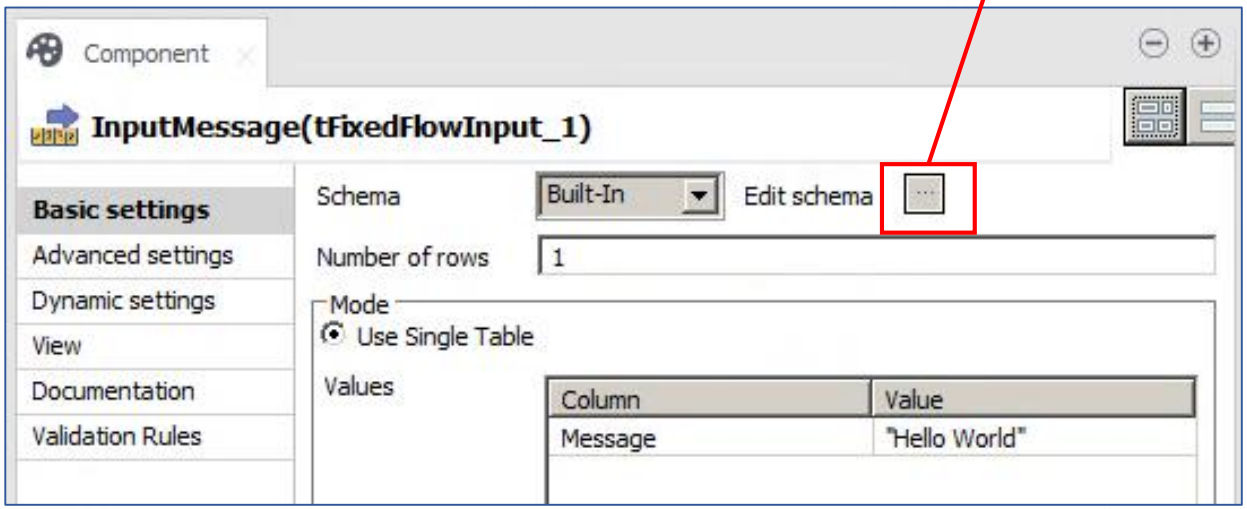


Build a Job

Configure components

- Use the **Component** View
- **Settings** are specific to each component





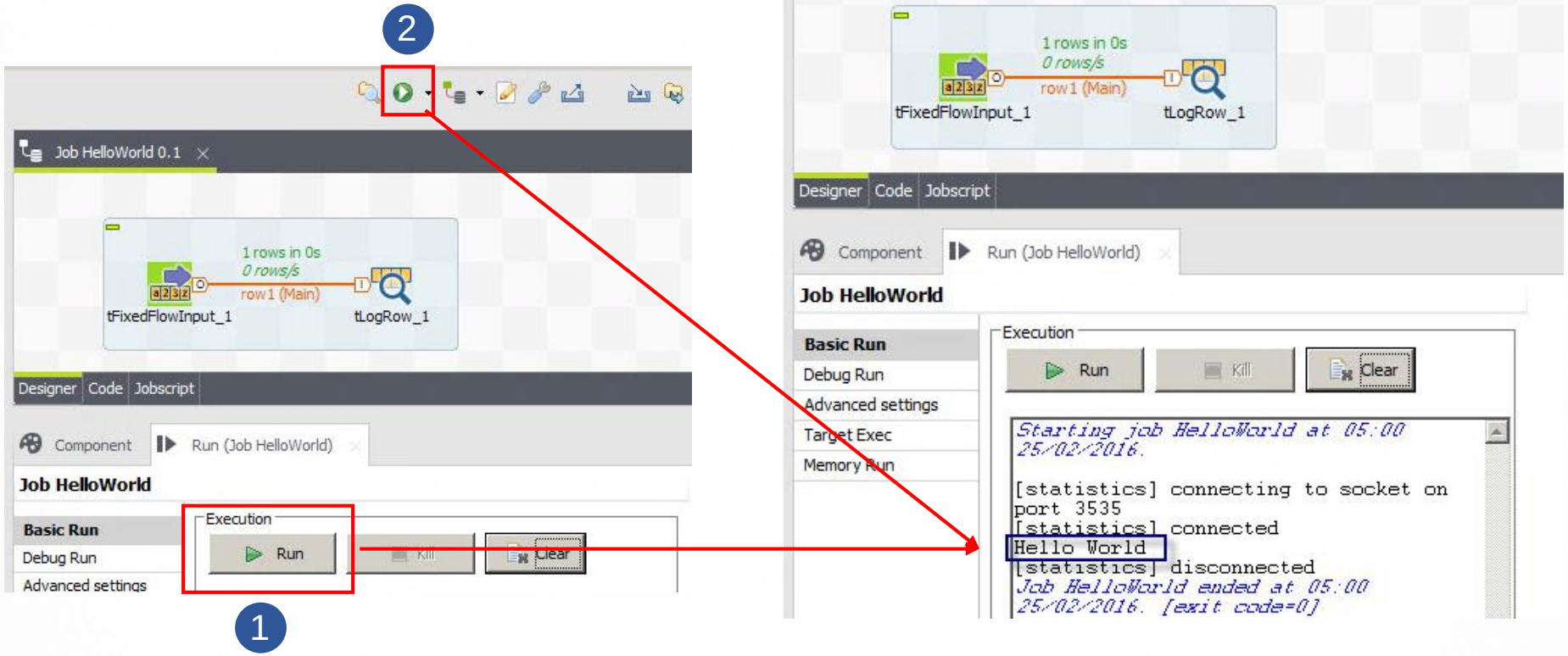
Example :

tFixedFlowInput settings

- Add a new Column to the Schema
- Define the value

Run a Job

- Two options
  1. Use the **Run View**
  2. Use quick access **Run button**



Lab overview

Discover Talend Studio and build a first Job to display a “Hello World” message

- Start Talend Studio
- Build the following Job

Lesson summary

- Talend Studio Main Window
  - Designer
  - Palette
  - Views
  - Repository
  - Quick Access Toolbar

- Build a Job in the **Designer** space
  - Use the **Palette** to find components
  - Configure components in the **Component View**
  - Use the **Run View** to explore Job's execution

Working with Files

Objectives

By the end of this lesson, you will be able to:

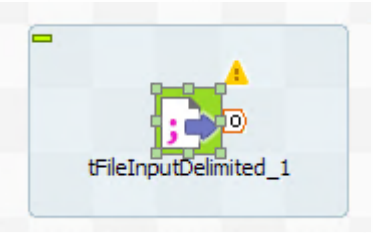
- Read and write a delimited text file
- Read warning and error messages
- Transform data using the tMap component
- Build simple expressions
- Explore data using the Data Viewer in the Studio
- Duplicate an existing Job as the basis for a new Job

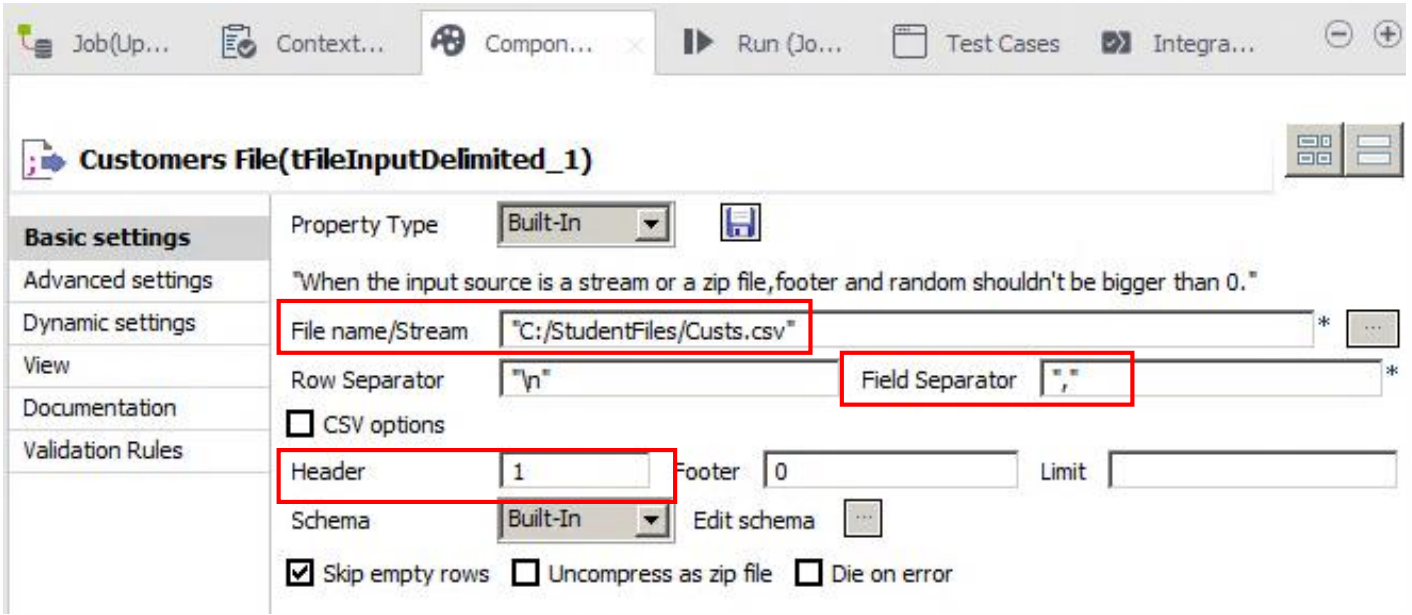
Use Case

- Apply basic transformations to an input file
  - Transform a column to uppercase
  - Concatenate first name and last name into full name

Read a Delimited File

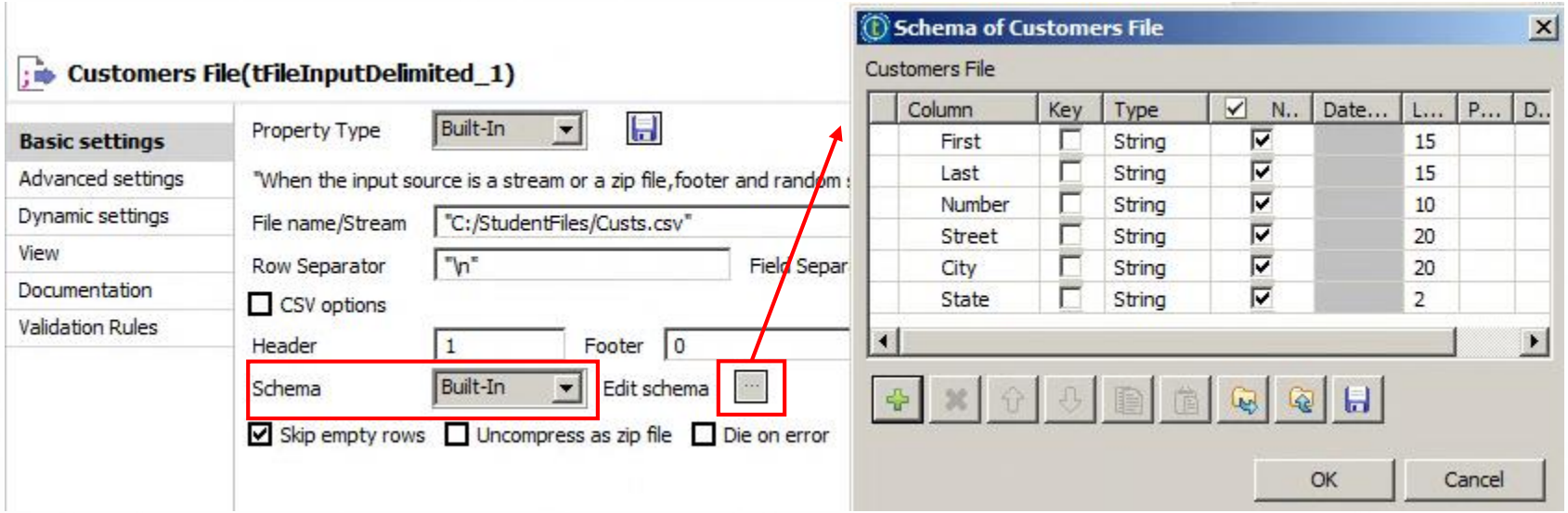
- Use the **tFileInputDelimited** component
  - Configure the file name
  - Check the field separator
  - Specify the number of header lines





File Delimited Schema

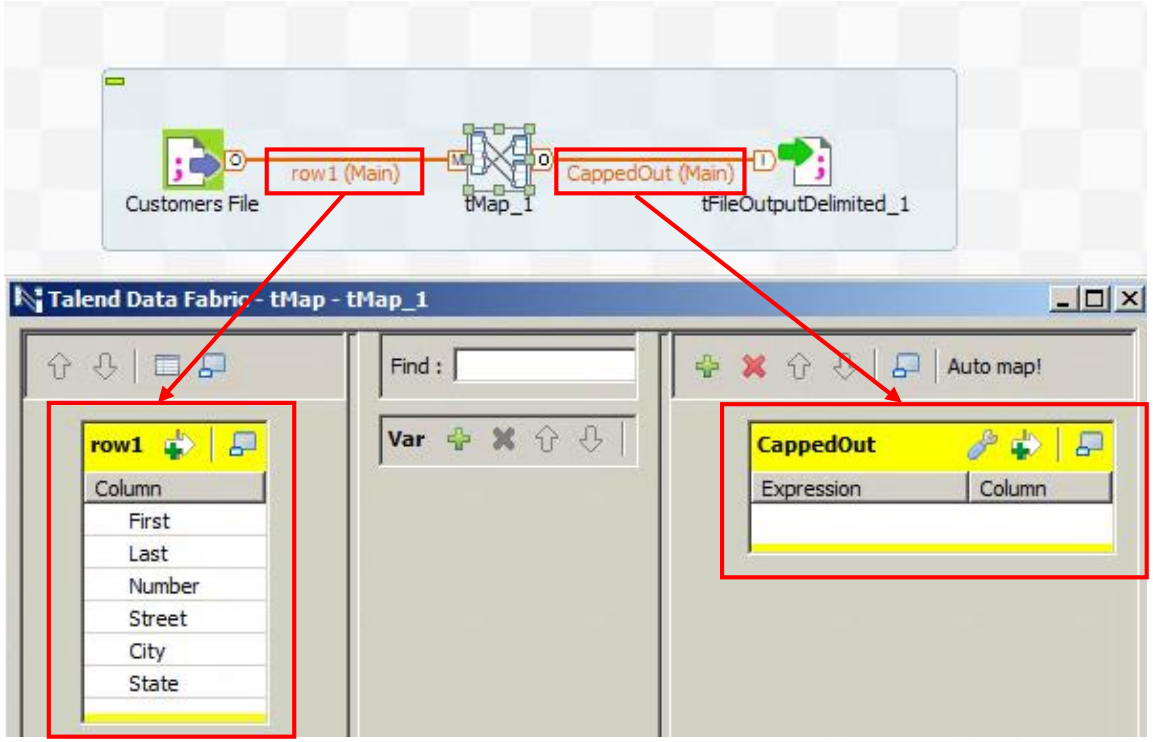
- The **Schema** defines the **structure** of the file
  - **Built-In** Type
    Schema defined **locally** for the selected component only (**tFileInputDelimited**)



Transforming Data

Input & Output Table

- Use a **tMap** component
  - Always between an input and an output component
    - **Table** name = name of the **Main Row**



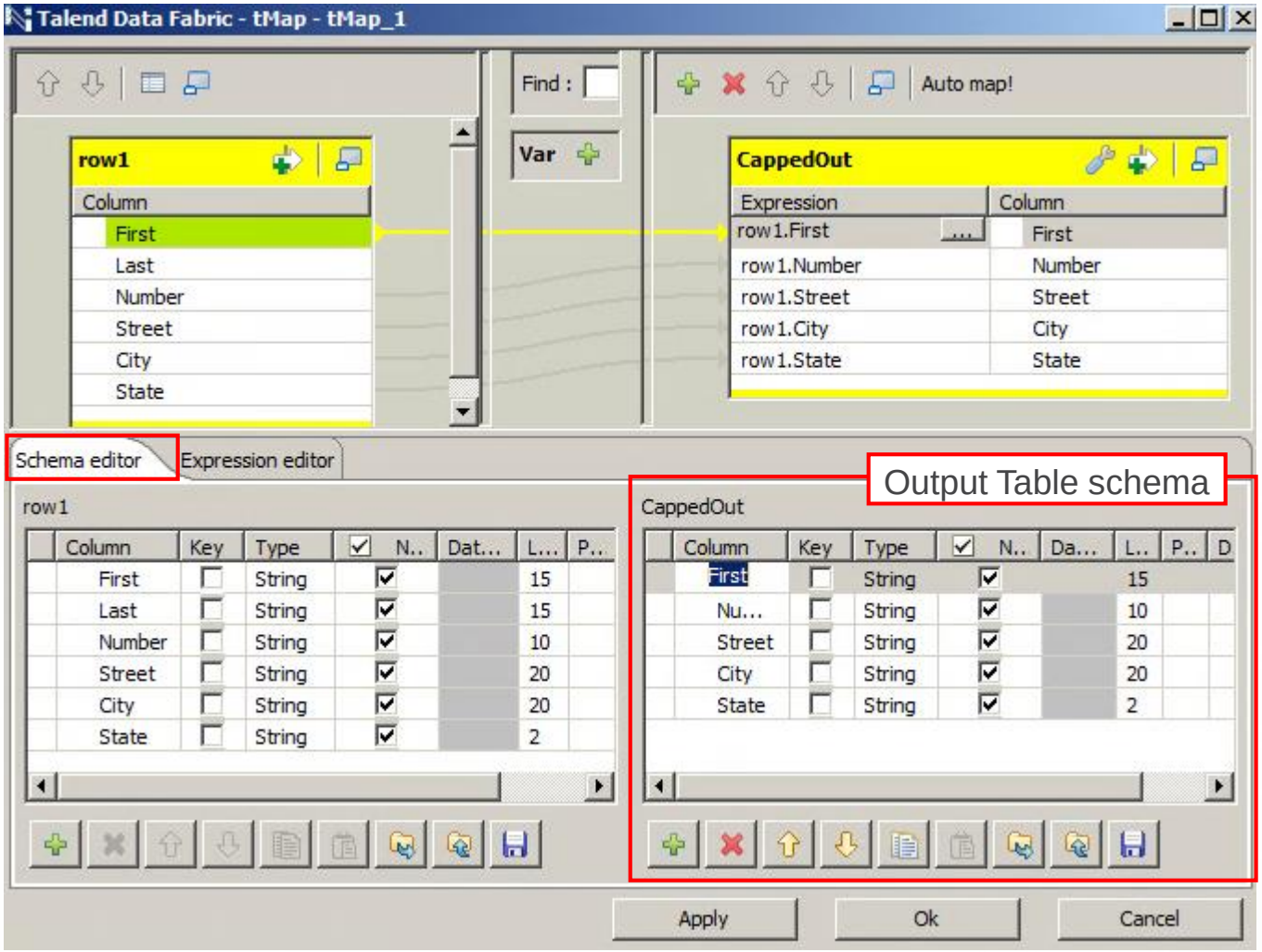
Input Table

Output Table

Transforming Data

Map rows

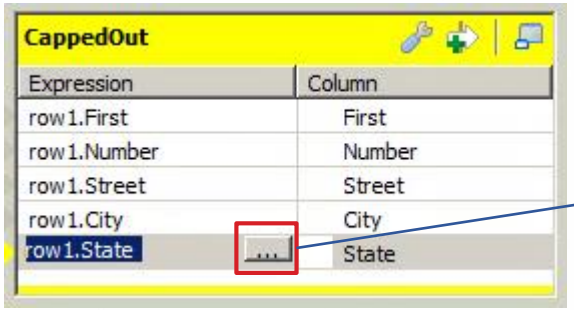
- Drag and drop input columns to the **Output Table**
- **Schema Editor –** allows schema updates

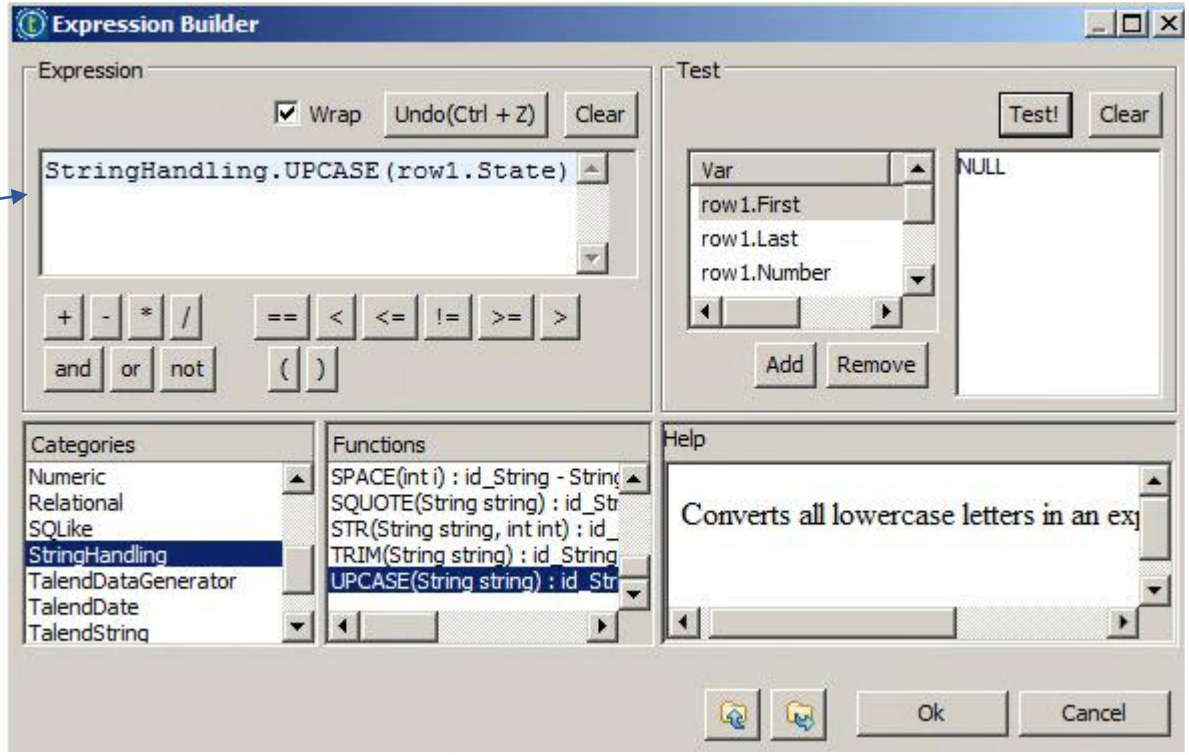


Transforming Data

Map Rows - Build Expressions

- Use pre-defined functions to build expressions

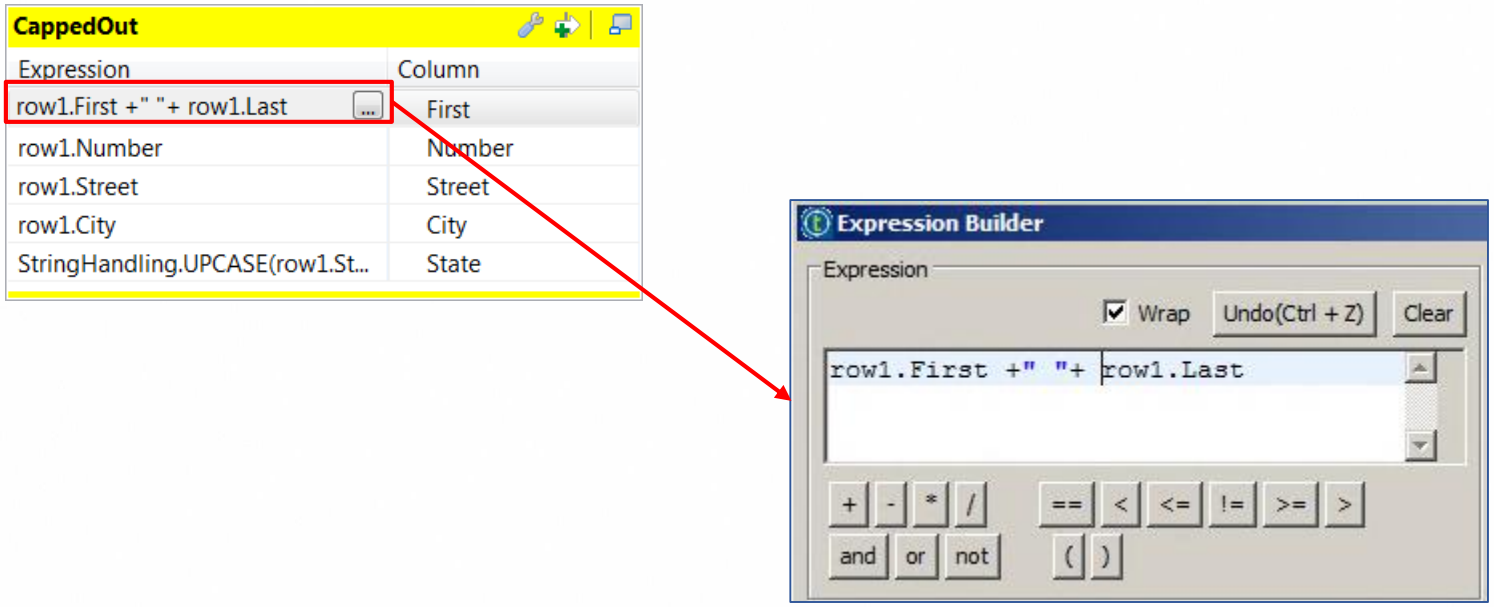




Transforming Data

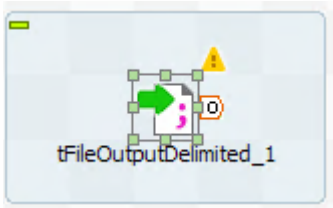
Map Rows - Build Expressions

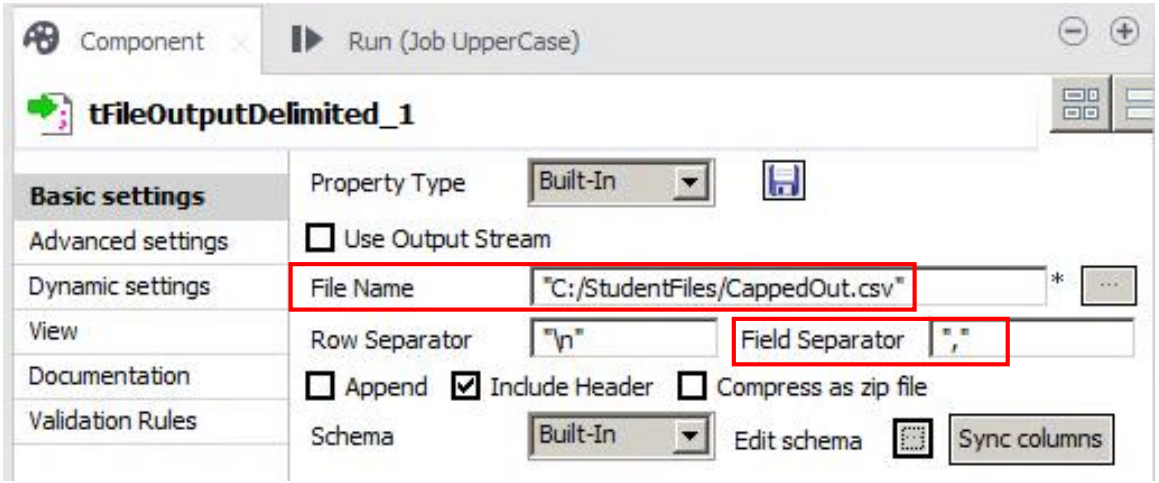
- Use operators and input columns to build expressions



Write to a Delimited File

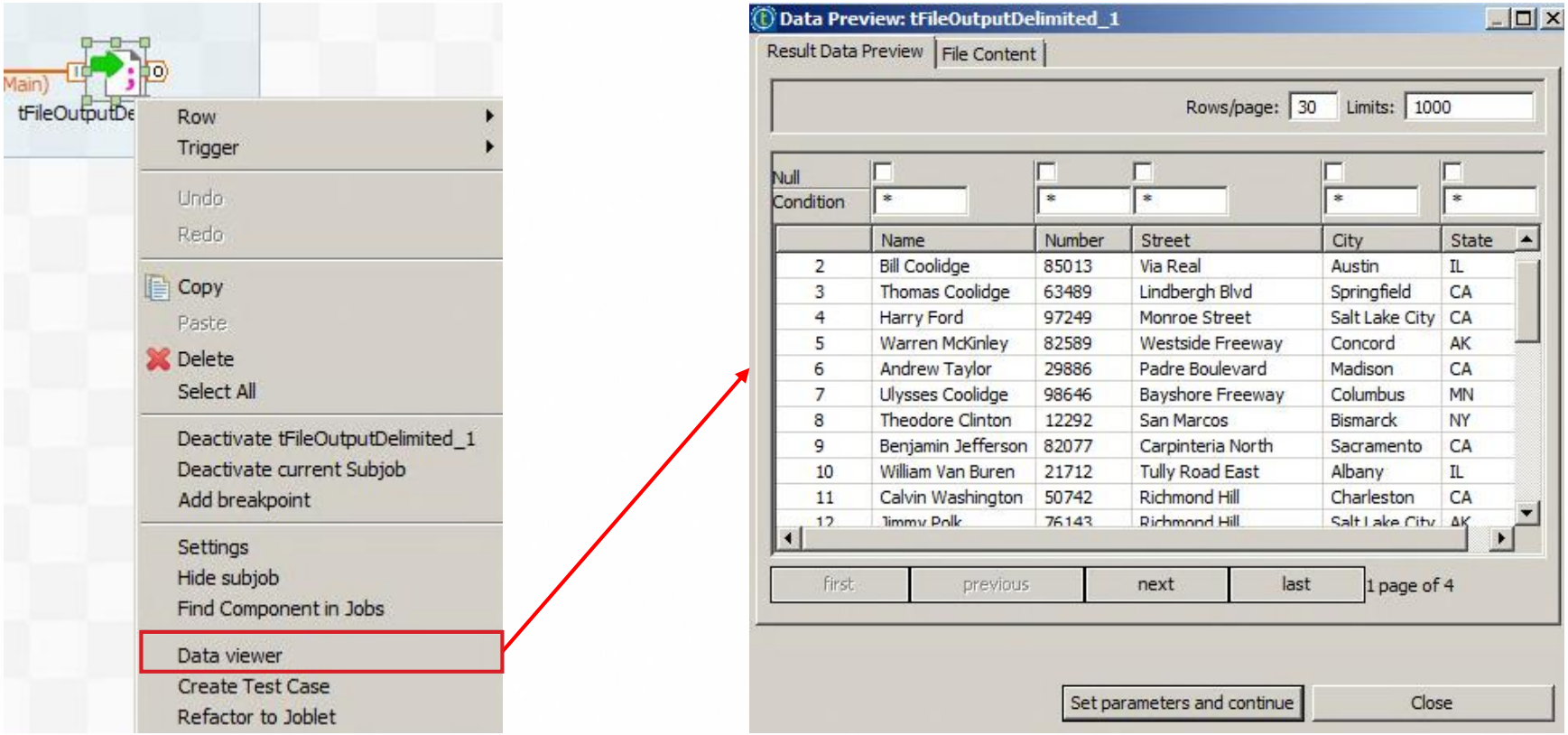
- Use **tFileOutputDelimited** component
  - Configure the file name
  - Check the field separator





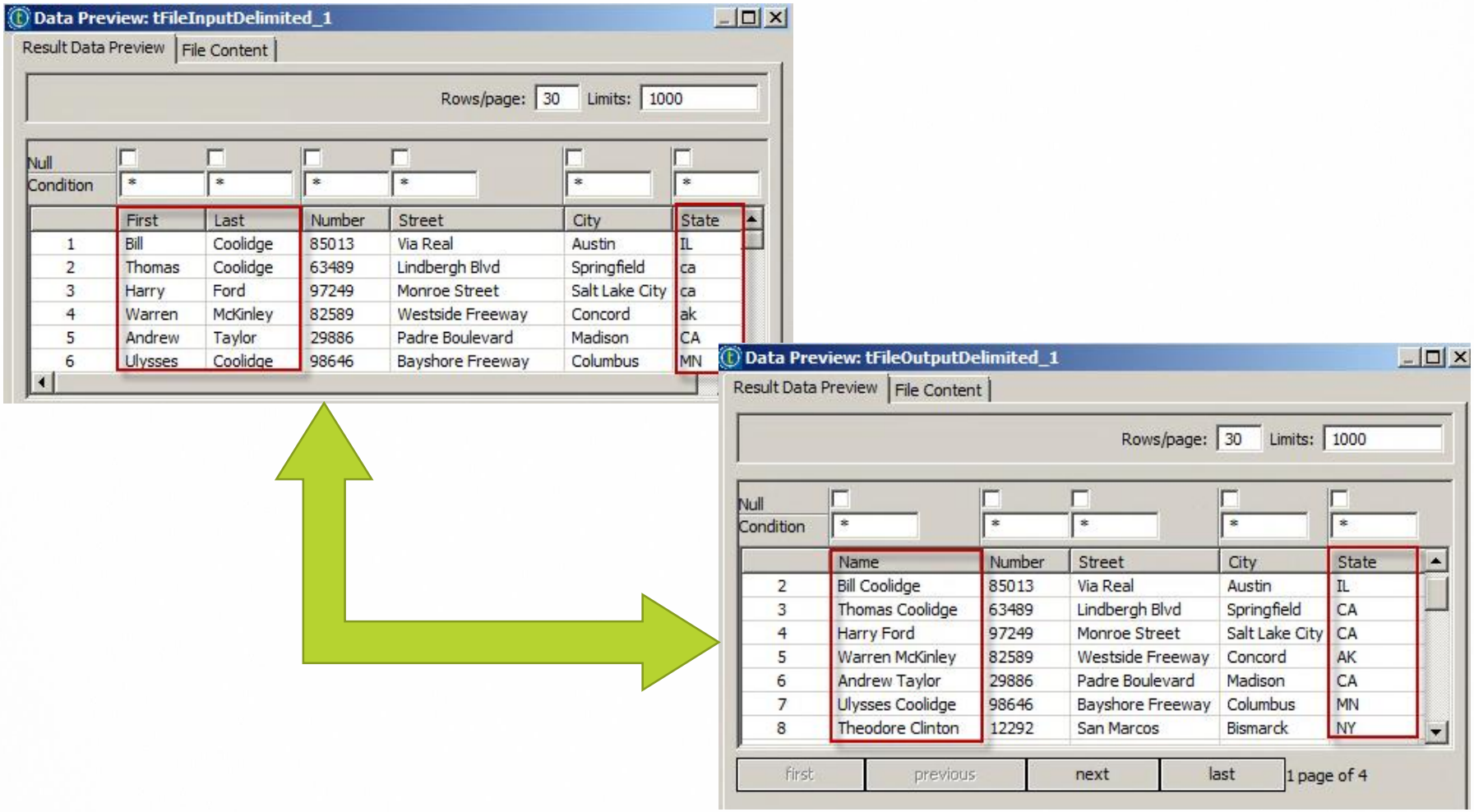
Data Viewer

- Available for both Input and Output components
- Allows quick access to data



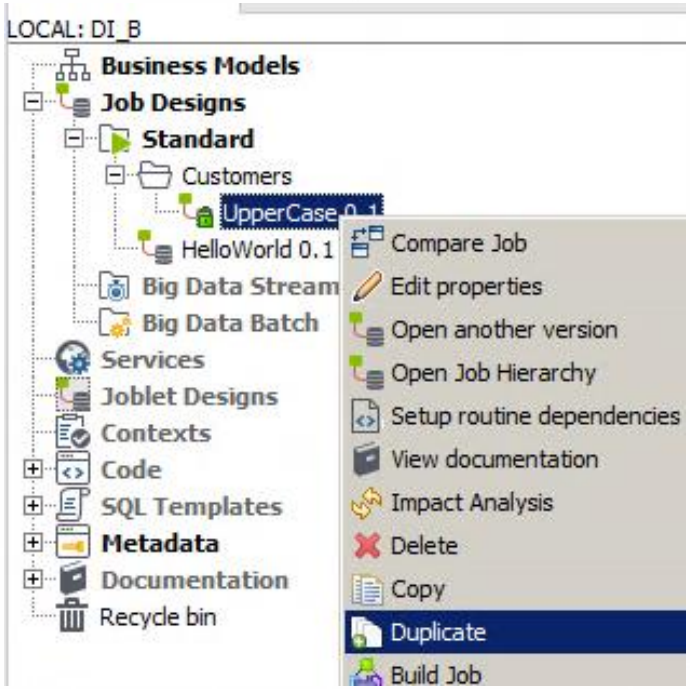
Transforming data

- Results



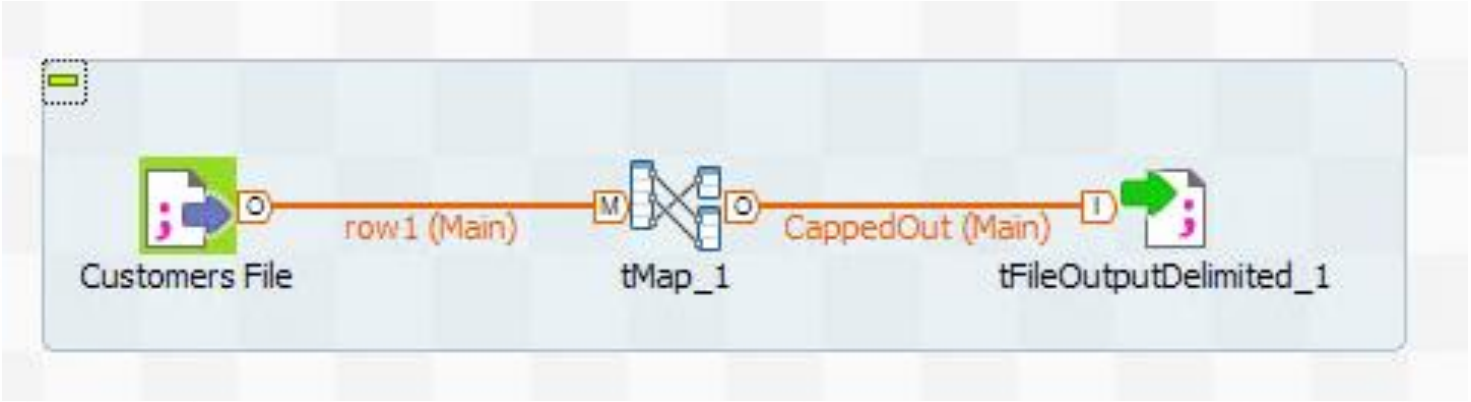
Duplicate a Job

- Allow reuse of simple Jobs in a more complex scenario
- Right-click the Job and select **Duplicate**



Lab overview

Build the following Job:



- Run Job and examine/confirm results
- Duplicate the Job

Lesson summary

- **tFileInputDelimited**
  - Reads a delimited file
- **tFileOutputDelimited**
  - Writes a delimited file
- **tMap**
  - Allows data transformation using operators and predefined functions

Joining Data Sources

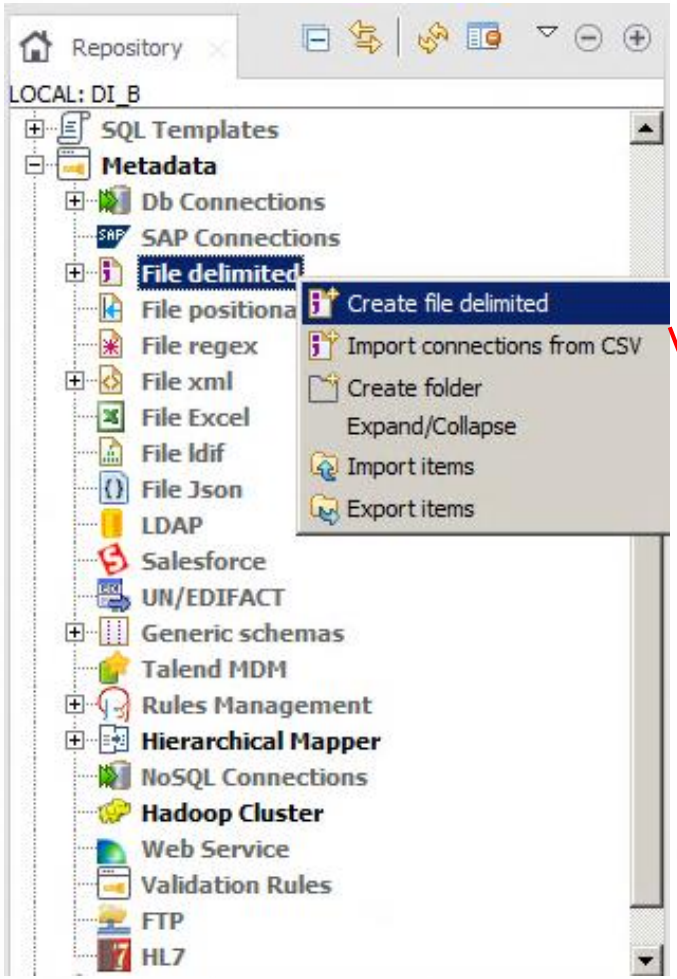
Multiple Data Sources

Lesson Objectives

By the end of this lesson, you will be able to:

- Store metadata centrally for use in other components/Jobs
- Use metadata
- Join two data sources
- Troubleshoot Join – handle rejects
- Log join rejects in the console

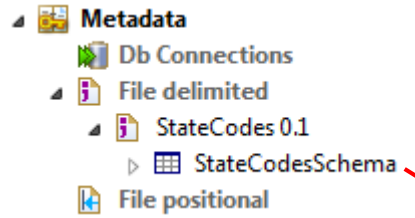
Create Metadata



- **Metadata**

  stores **configuration information** that can be **reused** in multiple components

- File Delimited - stores file information:
  - File Name
  - Separator & Header information
  - Schema (fields, types, length)



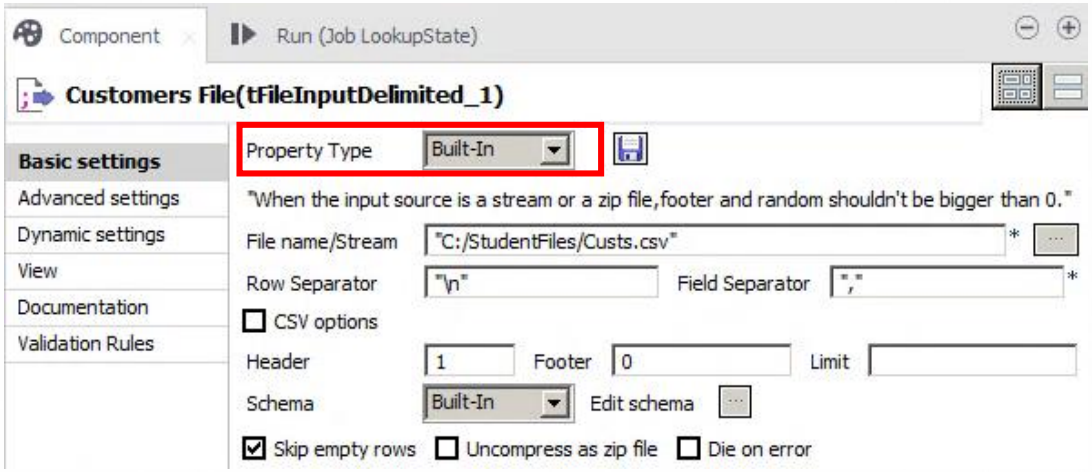
Description of the Schema

Column	Key	Type	N..	D...	Length
StateCode		String			2
StateName		String			20

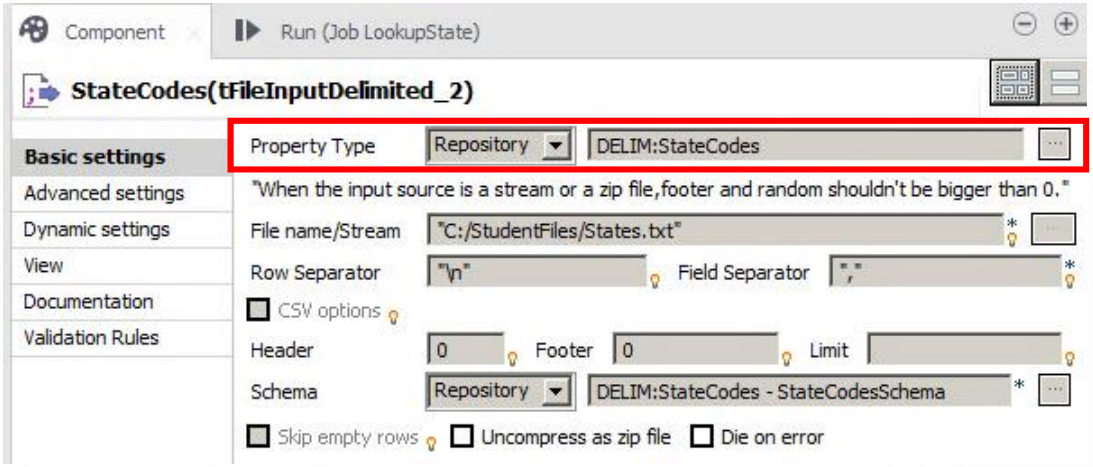
Use Metadata

Repository Schema Type

- Schema/Property types:
  - **Built-In**
    - Schema is defined **locally** for the components
    - Details are defined **locally**

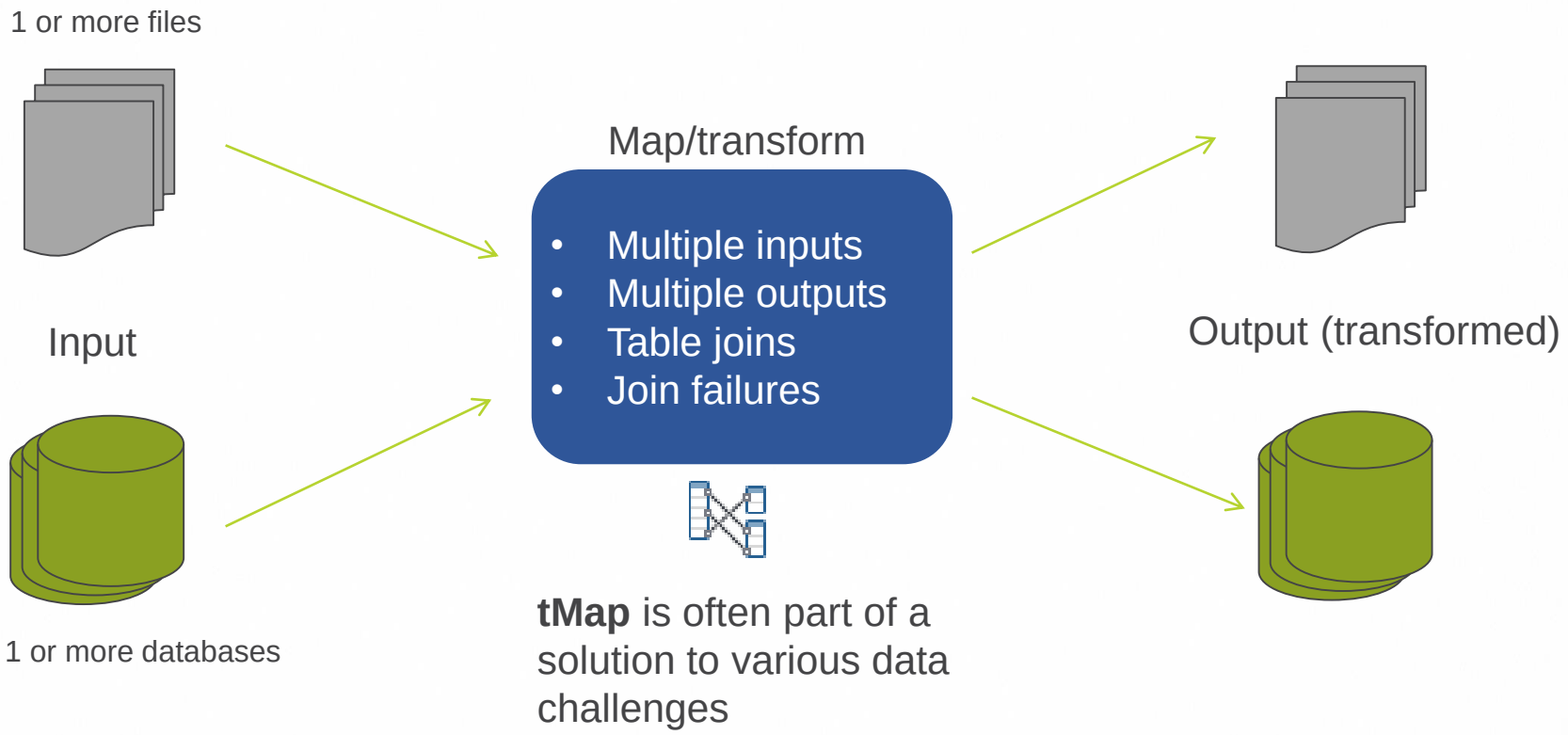


  - **Repository**
    - **Metadata** is referenced
    - Details are **predefined**



Joining Data Sources

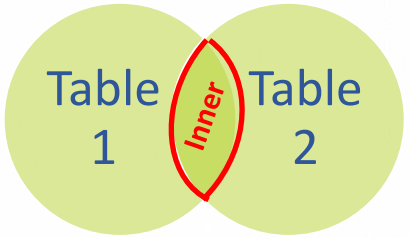
Use tMap to join and transform multiple inputs to multiple outputs



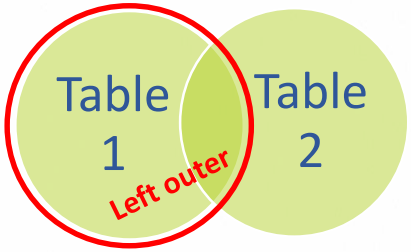
Joining Data Sources

Join Types

- Join - Combines rows from two or more tables
  - **Inner Join**
    - Returns rows that are in *both* tables



  - **Left Outer Join** (or left join)
    - Returns *all records* from the left table (Table 1) even if a match is not found in the right table (Table 2)… but with NULL in the column for right table



Joining Data sources

Inner Join or Left Outer Join?

Id	Name	Gender
1	Shong	M
2	Ross	M
3	Sabrina	F
4	Elisa	F

Inner Join

Left Outer Join

User_id	Name	Country	Email
1	Shong	CN	shong@talend.com
4	Elisa	FR	elisa@talend.com

Id	Name	Email
1	Shong	shong@talend.com
4	Elisa	elisa@talend.com

Id	Name	Email
1	Shong	shong@talend.com
2	Ross	null
3	Sabrina	null
4	Elisa	elisa@talend.com

Join data using tMap

Define Multiple Input Sources

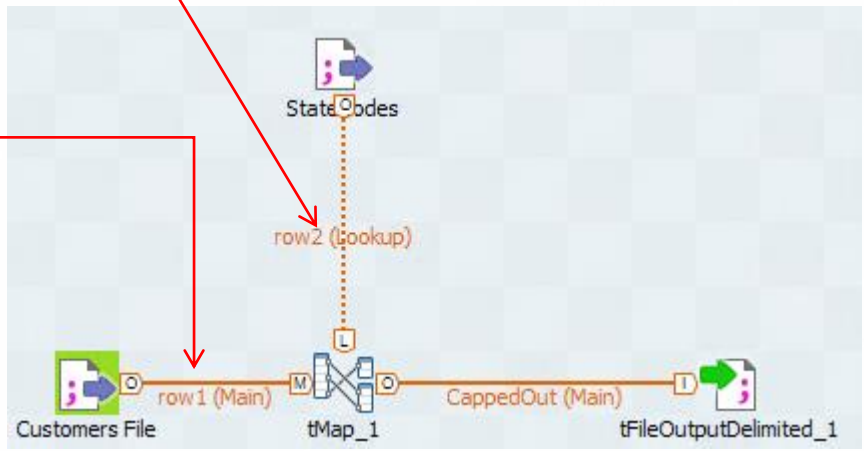
Configure Join

Define Multiple Output Tables

Handle Rejects

Display Rejects

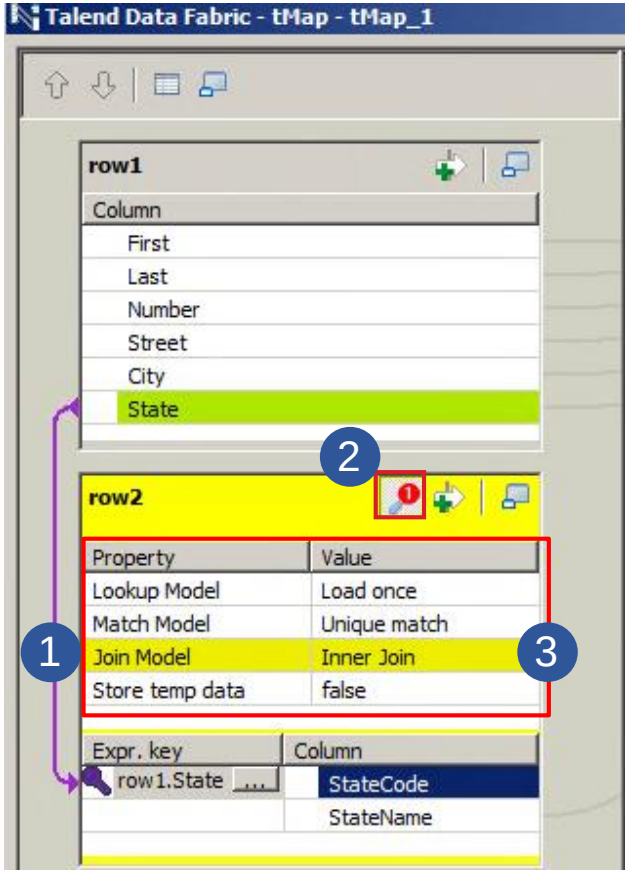
- **tMap** accepts multiple input sources to join
  - **Main** flow (only **one** can be Main row)
  - **Lookup** flow (the rest of the input rows)



Join data using tMap

- Define Multiple Input Sources
- **Configure Join**
- Define Multiple Output Tables
- Handle Rejects
- Display Rejects

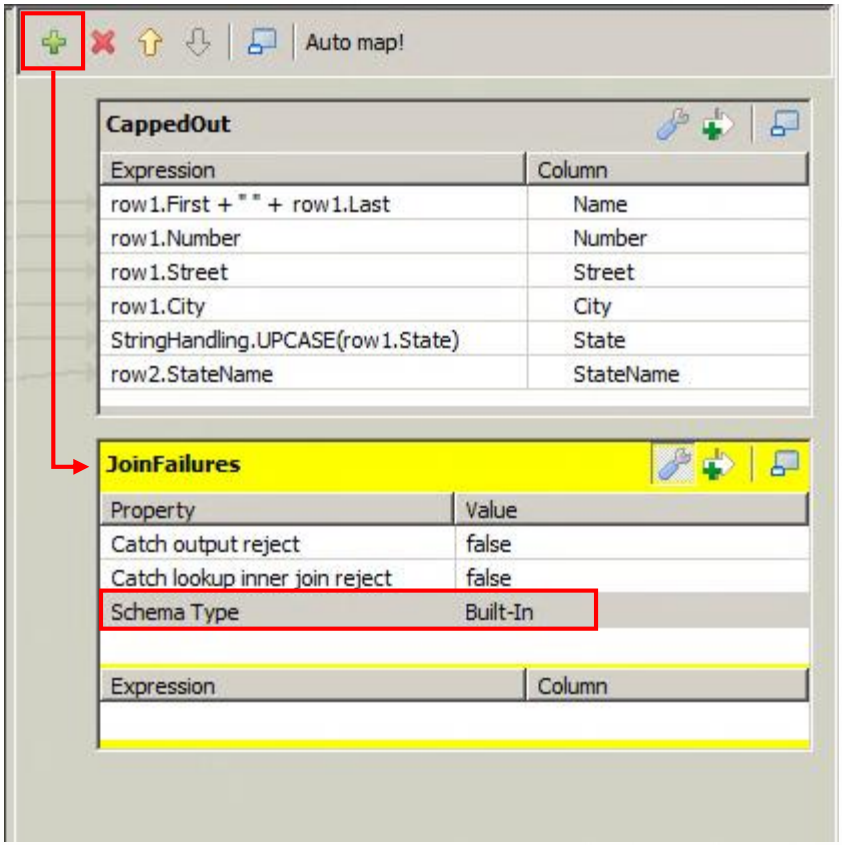
1. Join data
2. Expand Model properties
3. Define Join Model: **Inner Join** or **Left Outer Join**



Join data using tMap

- Define Multiple Input Sources
- Configure Join
- **Define Multiple Output Tables**
- Handle Rejects
- Display Rejects

Add Output Table



Output table Schema Type can be:

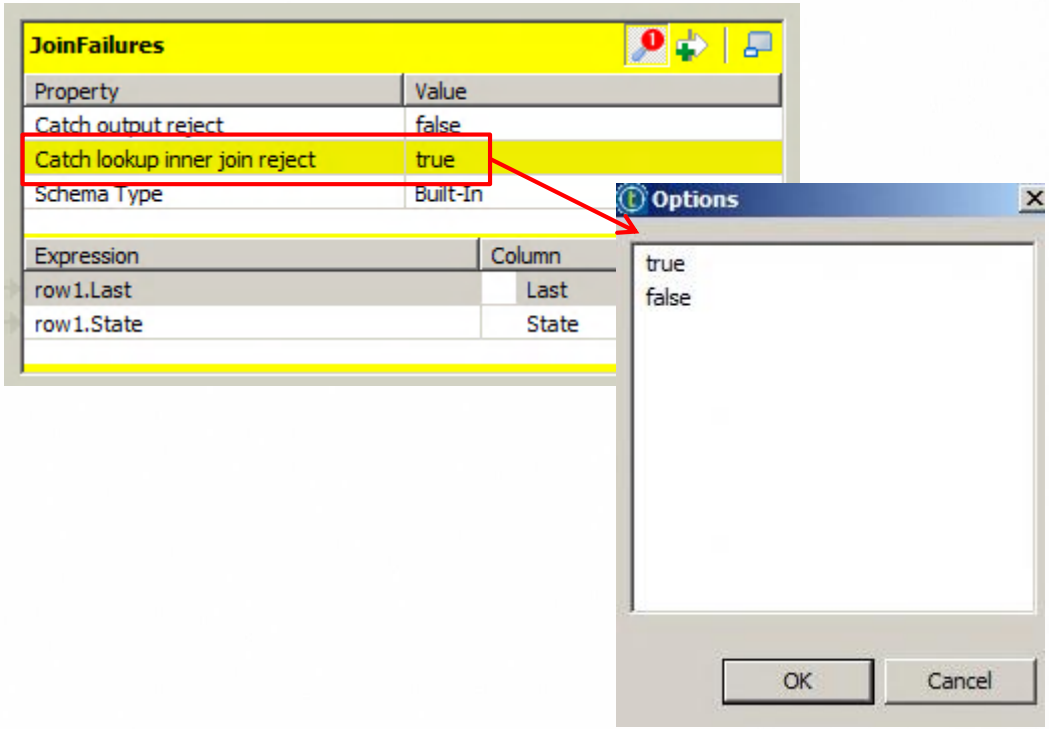
- Built-in
- Repository (reference metadata)

Join data using tMap

- Define Multiple Input Sources
- Configure Join
- Define Multiple Output Tables
- **Handle Rejects**
- Display Rejects

- **Inner Join Rejects** can be automatically caught using the following property:

Example failures/rejects output table



Join data using tMap

Define Multiple Input Sources

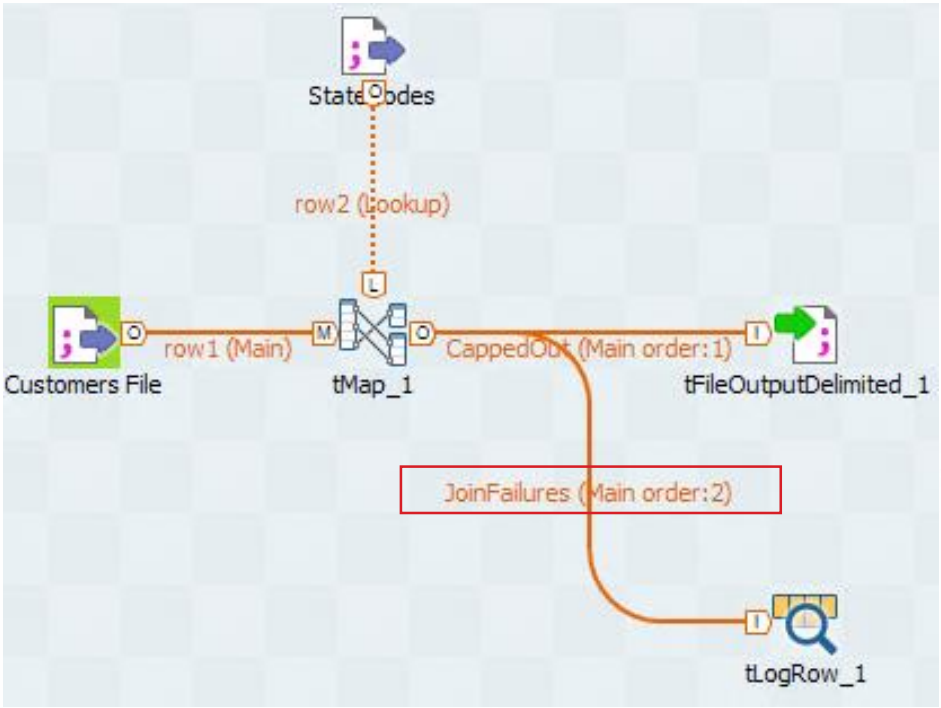
Configure Join

Define Multiple Output Tables

Handle Rejects

Display Rejects

- **Inner Join Rejects** can be displayed in a separate output component

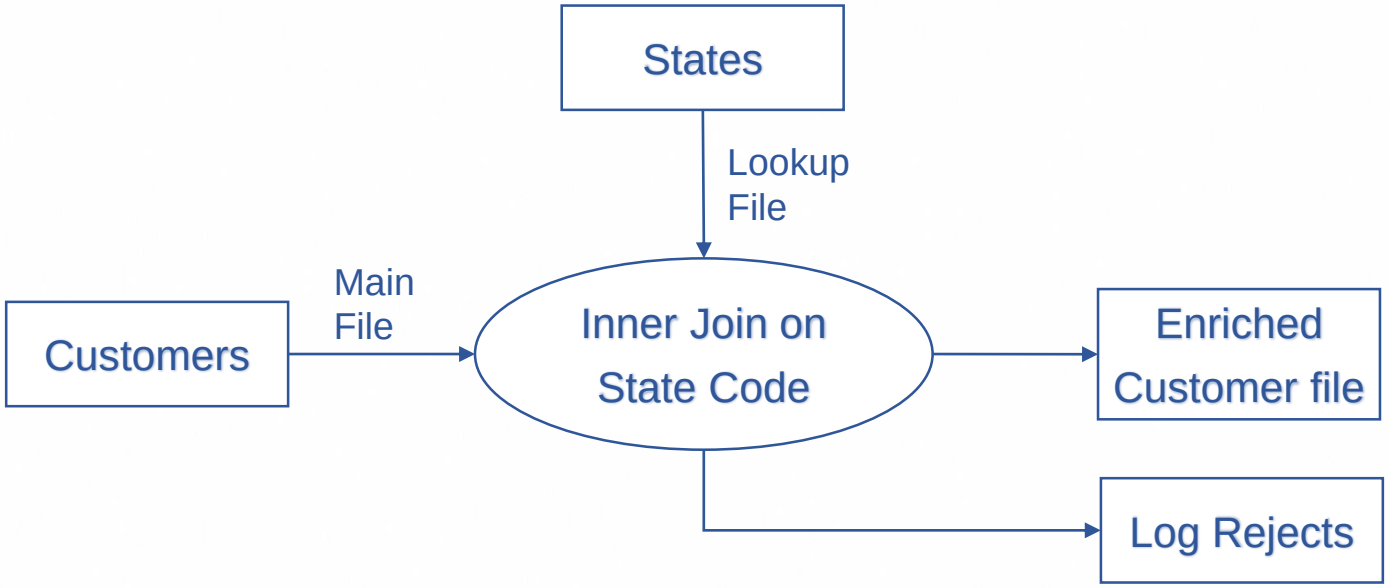


Lab overview

Joining multiple data sources

Enrich customer data file with state names coming from an external file:

- Create file delimited repository metadata for states file
- Build the following Job:



- Run Job and examine/confirm results

Lesson summary

Metadata

- stores configuration information that can be reused
- Referenced by **Repository** Schema type

tMap includes extensive functionality, such as:

- **Multiple input** flows (from multiple sources)
- Can **Join** data sources (inner, left outer)
- Can perform **complex transformations**
- Catches **rejects**
- Output tables accept both **Built-In & Repository** Schemas

Filtering Data

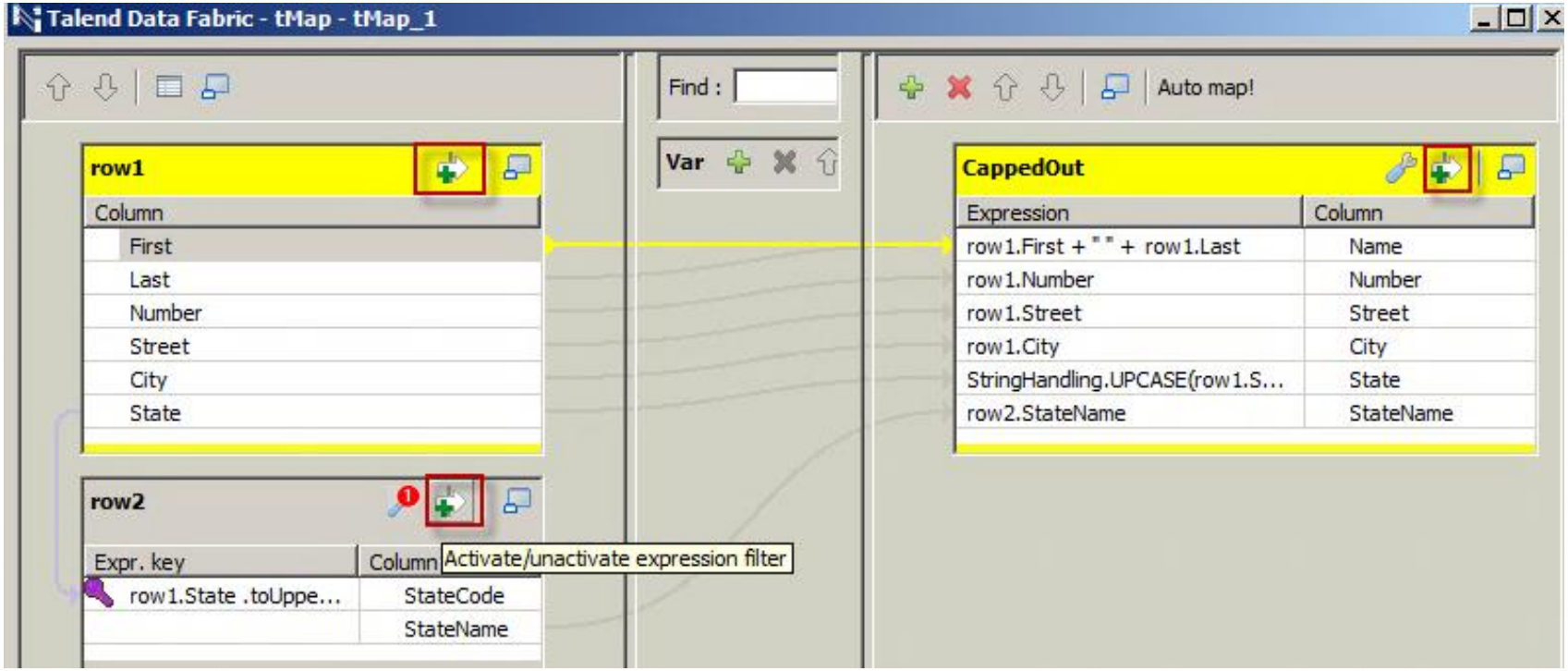
Objectives

By the end of this lesson, you will be able to:

- Use the tMap component to filter data
- Execute Job sections conditionally
- Duplicate output flows

Filtering Data

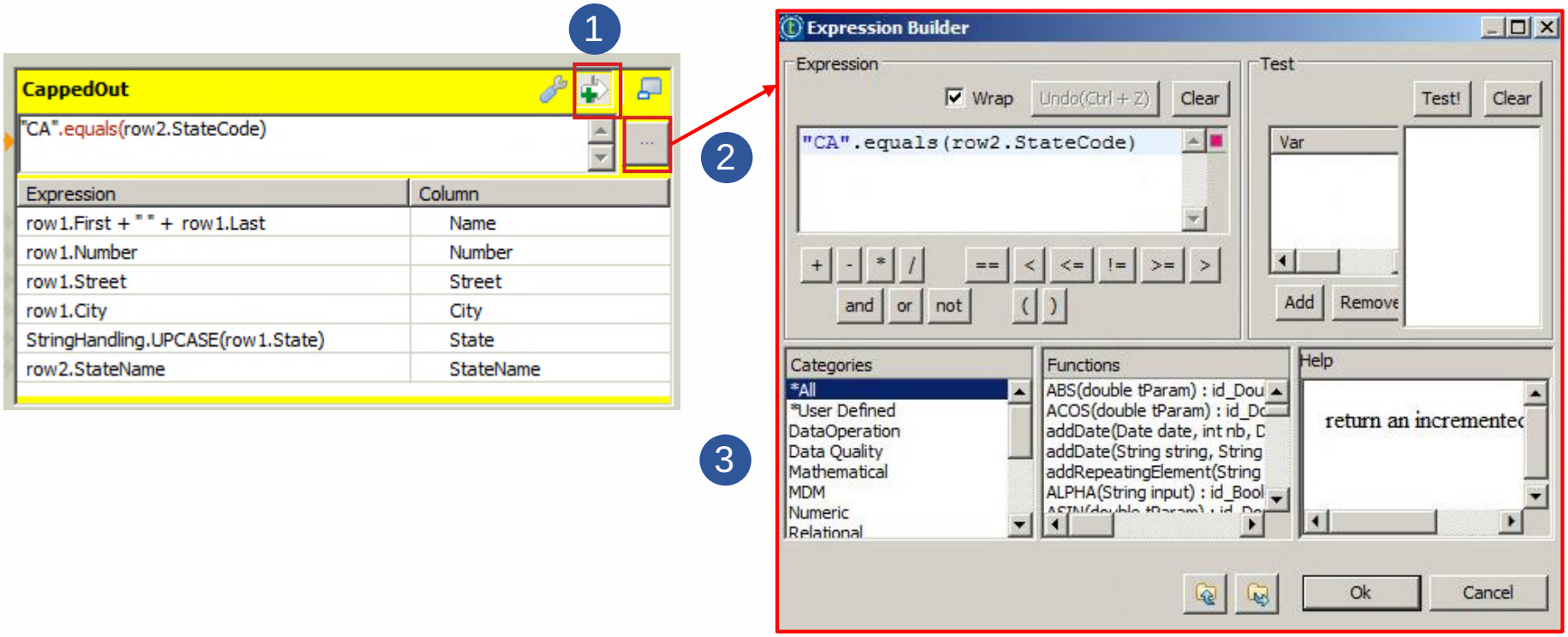
- Retrieve a limited set of data based on conditions
- The **tMap** component allows data filtering
  - Filters can be applied on any **input** or **output** table



Configure Data Filtering

Add a Filter

1. Activate the filtering option.
2. Set the filter expression
3. Use the **Expression Builder** to assist if needed



Filtering Data

Example filtering column data

Input Data

	First	Last	Number	Street	City	State
1	Bill	Coolidge	85013	Via Real	Austin	IL
2	Thomas	Coolidge	63489	Lindbergh Blvd	Springfield	ca
3	Harry	Ford	97249	Monroe Street	Salt Lake City	ca
4	Warren	McKinley	82589	Westside Freeway	Concord	ak
5	Andrew	Taylor	29886	Padre Boulevard	Madison	CA
6	Ulysses	Coolidge	98646	Bayshore Freeway	Columbus	MN
7	Theodore	Clinton	12292	San Marcos	Bismarck	NY
8	Benjamin	Jefferson	82077	Carpinteria North	Sacramento	ca
9	William	Van Buren	21712	Tully Road East	Albany	IL
10	Calvin	Washington	50742	Richmond Hill	Charleston	ca
11	Jimmy	Polk	76143	Richmond Hill	Salt Lake City	AK
12	Calvin	Adams	52386	Lake Tahoe Blvd.	Montgomery	NY
13	Ulysses	Monroe	70511	Jones Road	Trenton	IL
14	Zachary	Tyler	45040	Santa Rosa North	Carson City	AK
15	Ulysses	Johnson	19989	Via Real	Juneau	AL
16	George	Arthur	89874	Calle Real	Annapolis	AL
17	George	Jefferson	67703	Fontaine Road	Pierre	IL
18	Herbert	Grant	90635	North Ventu Park Road	Columbus	AK
19	Calvin	Washington	37446	E Fowler Avenue	Pierre	al
20	John	Harrison	86745	San Marcos	Annapolis	AK

Filtered – California only
Transformed – Upper case

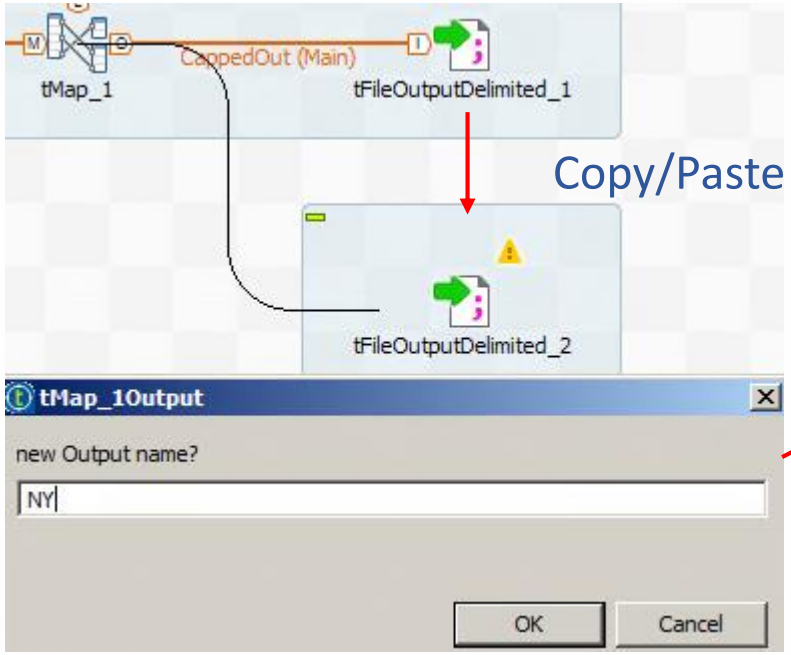
Filtered & Transformed Data

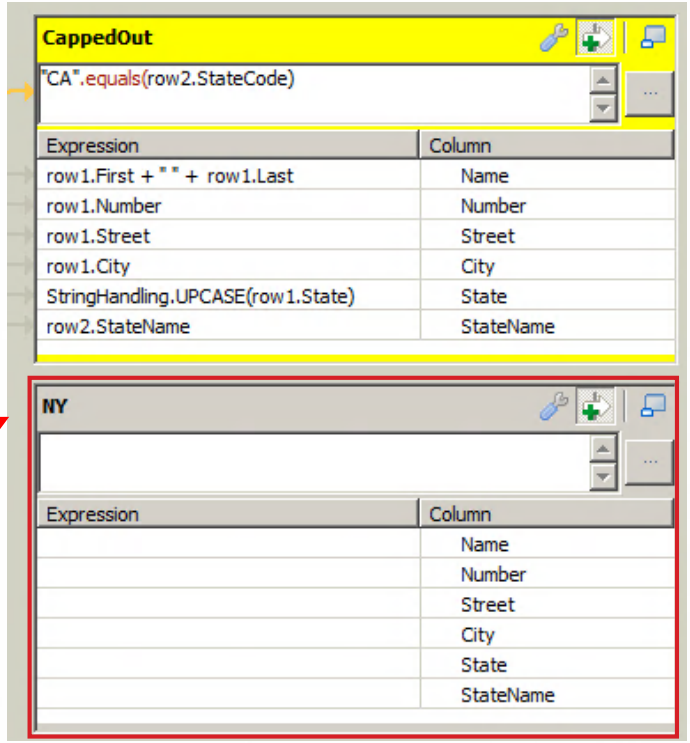
	Name	Number	Street	City	State
1	Thomas Coolidge	63489	Lindbergh Blvd	Springfield	CA
2	Harry Ford	97249	Monroe Street	Salt Lake City	CA
3	Andrew Taylor	29886	Padre Boulevard	Madison	CA
4	Benjamin Jefferson	82077	Carpinteria North	Sacramento	CA
5	Calvin Washington	50742	Richmond Hill	Charleston	CA

Duplicate Output Flows

Duplicate Component

- Copy/Paste output component
- Create connection with **tMap**
  - Schema is propagated to the **tMap** component
  - Mapping configuration is **empty** by default

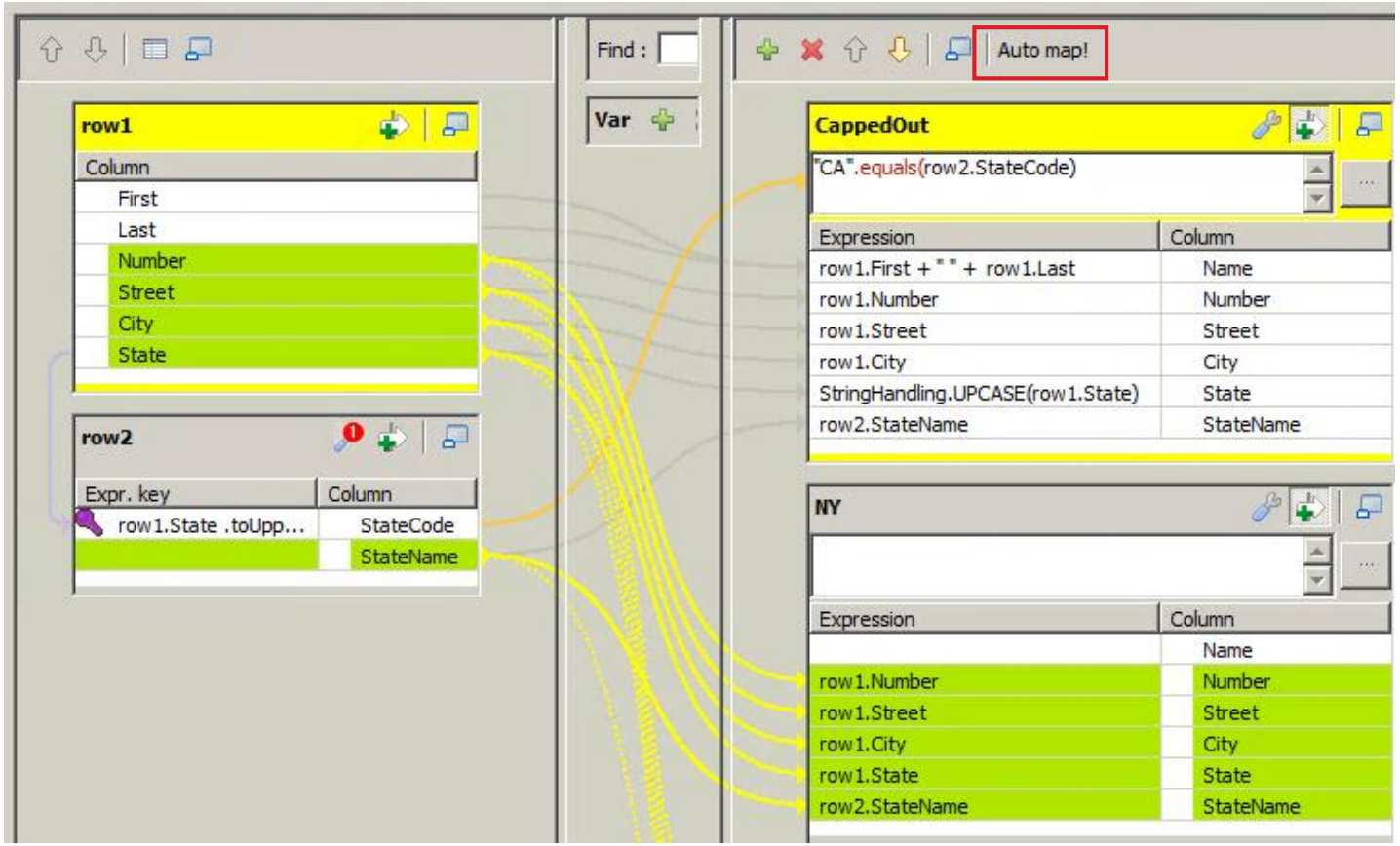




Duplicate Output Flows

Use Auto Map to duplicate output flows within tMap

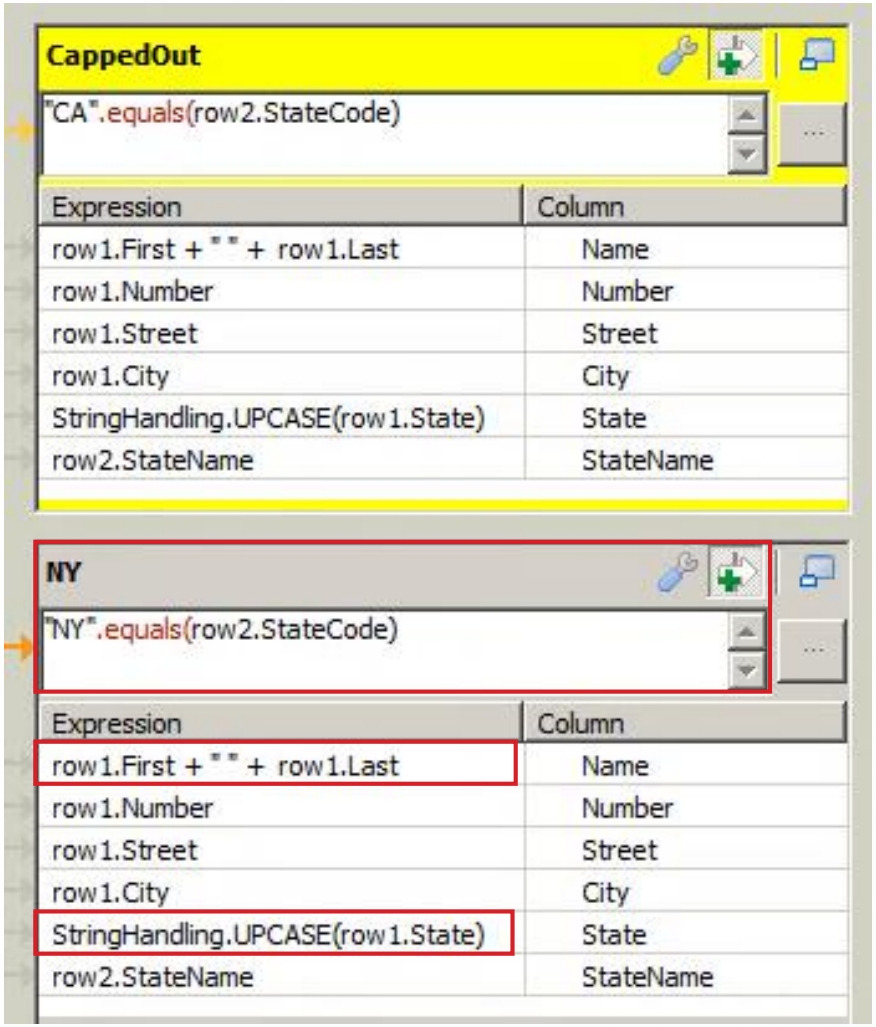
- The **Auto map!** function:
  - -Maps **input** columns **with the same name** (highlighted in green)



Duplicate Output Flows

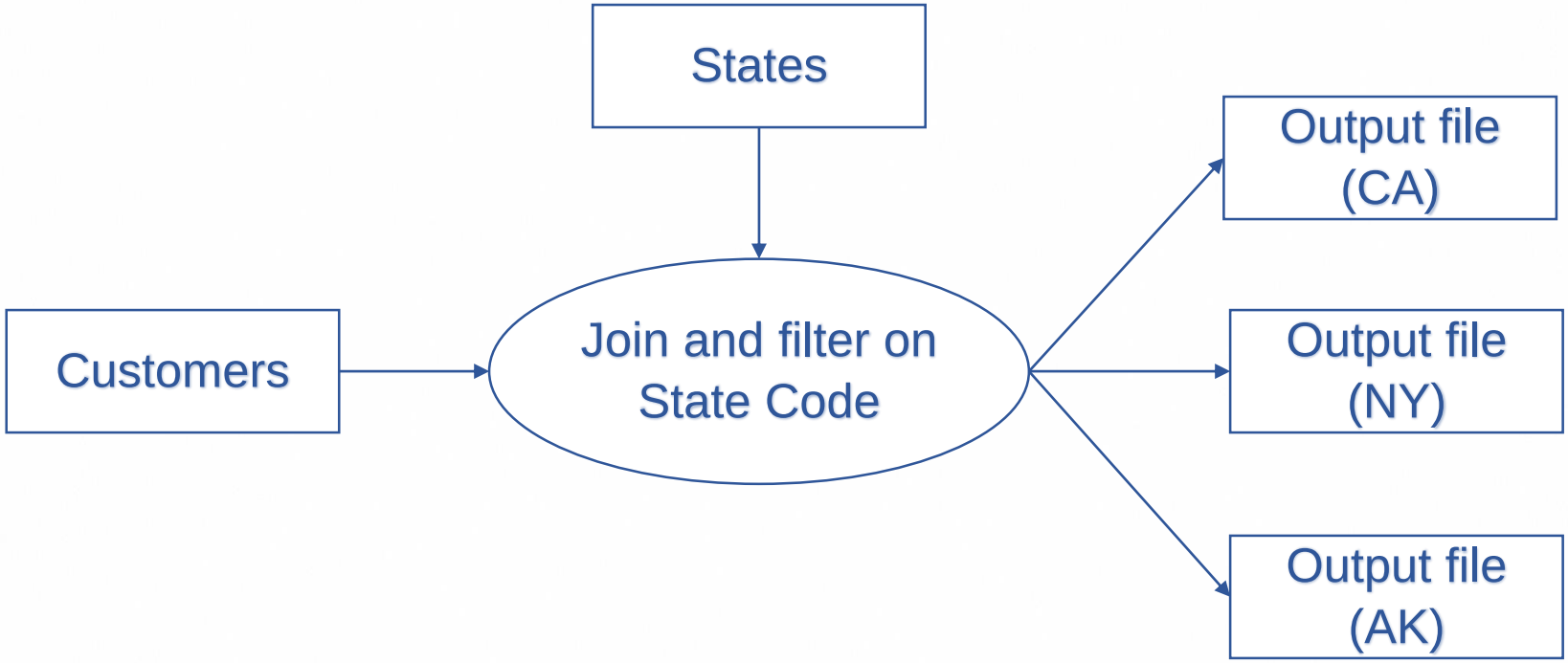
Finish tMap configuration

- Configure the new output table
  - Add filter
  - Apply transformation rules



Lab overview

- Build a Job that generates three different output files (by filtering input data per state):



- Run Job and examine output files

Lesson summary

tMap

- Allows **data filtering** in input and output tables
- Allows **Multiple output** flows (to files, databases, …)

Remember that **tMap** also allows

- Multiple Input flows
- Complex Data transformation
- Joining data
- Catching rejects

Using Context Variables

Objectives

By the end of this lesson, you will be able to:

- Create Built-in Context Variables, specific for a Job
- Run a Job using Built-in Context Variables
- Use Context Variables
- Create Repository Context Variables, available for all Jobs
- Run a Job using Repository Context Variables

What is a Context Variable?

- **Context Variable**s
  - Can take different values depending on the context

Context Variable	Development	Test	Production
InputDir	C:/Dev/Input	C:/Test/Input	C:/Prod/Input
OutputDir	C:/Dev/Output	C:/Test/Output	C:/Prod/Output

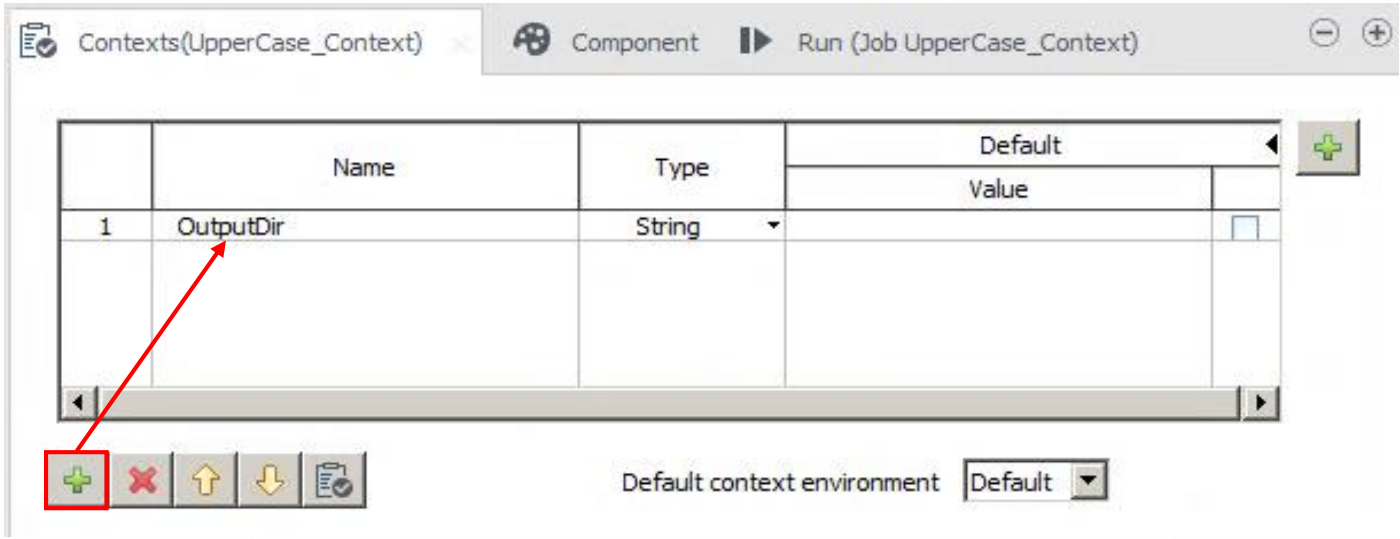
- A Job is always run in a **specific context**
- **Goal:** Specify the **value to use** by changing the **context** in which you run your Job

Define Context Variables

Context View

- **Add variable(s)**
- Add context(s)
- Set default context environment
- Set pre-defined values

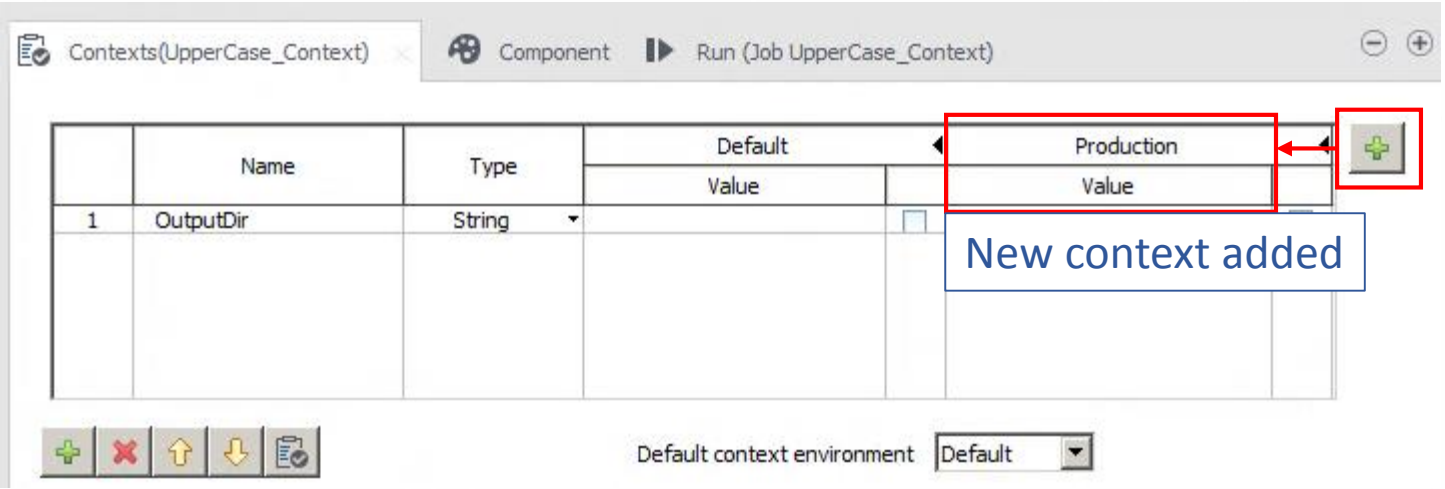
Context View is available for each Job



Define Context Variables

Context View

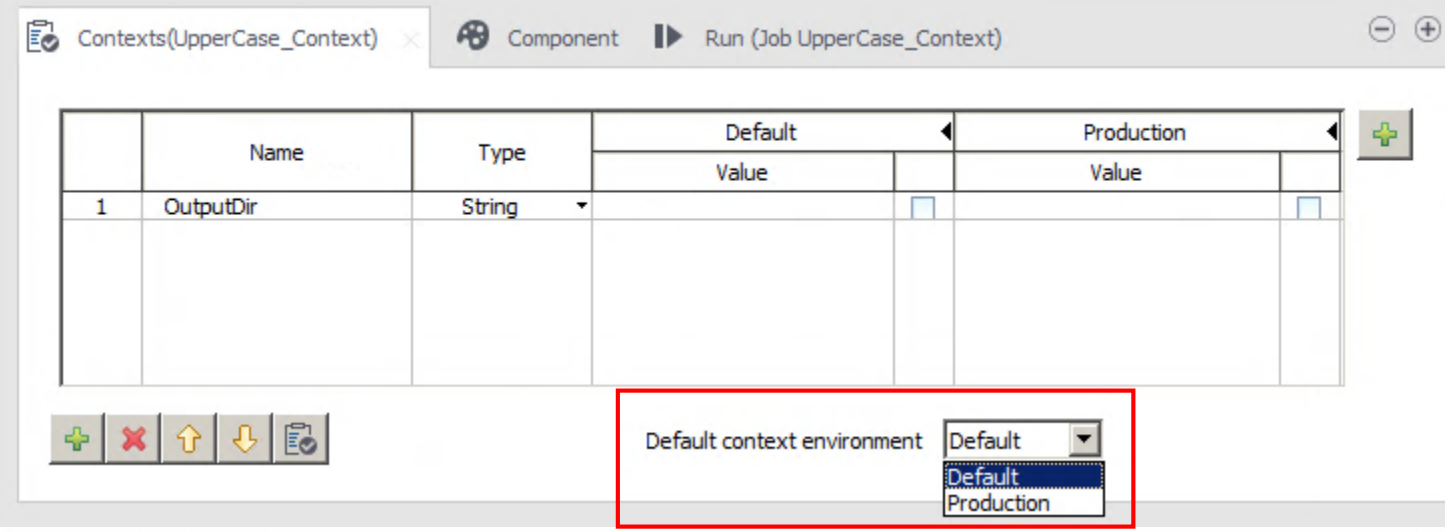
- Add variable(s)
- **Add context(s)**
- Set default context environment
- Set pre-defined values



Define Context Variables

Context View

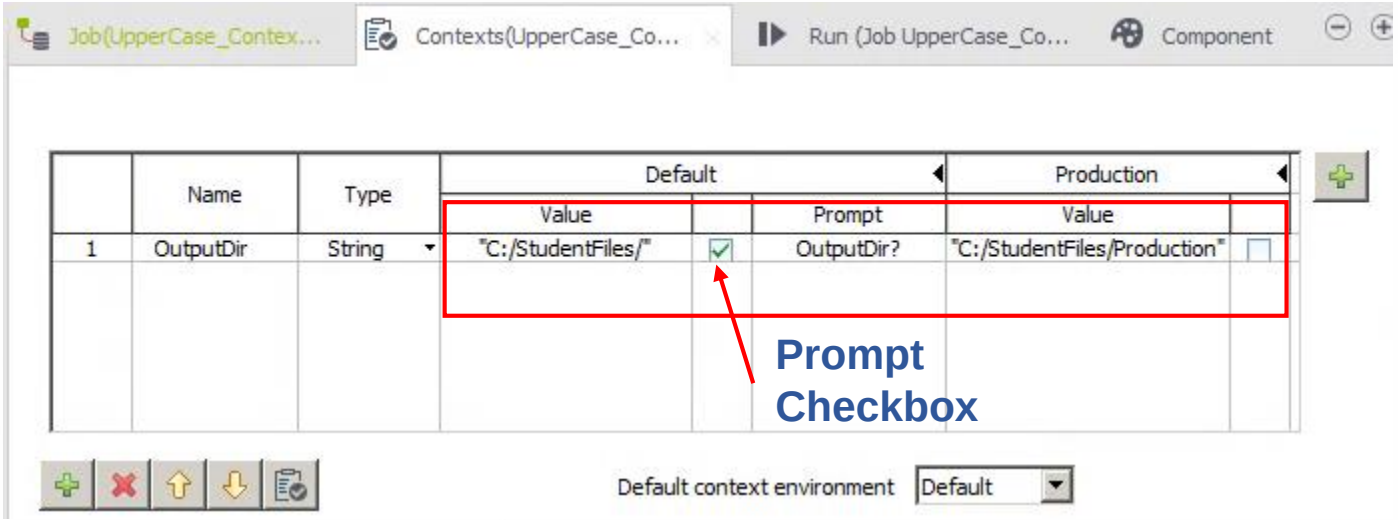
- Add variable(s)
- Add context(s)
- **Set default context environment**
- Set pre-defined values



Define Context Variables

Context View

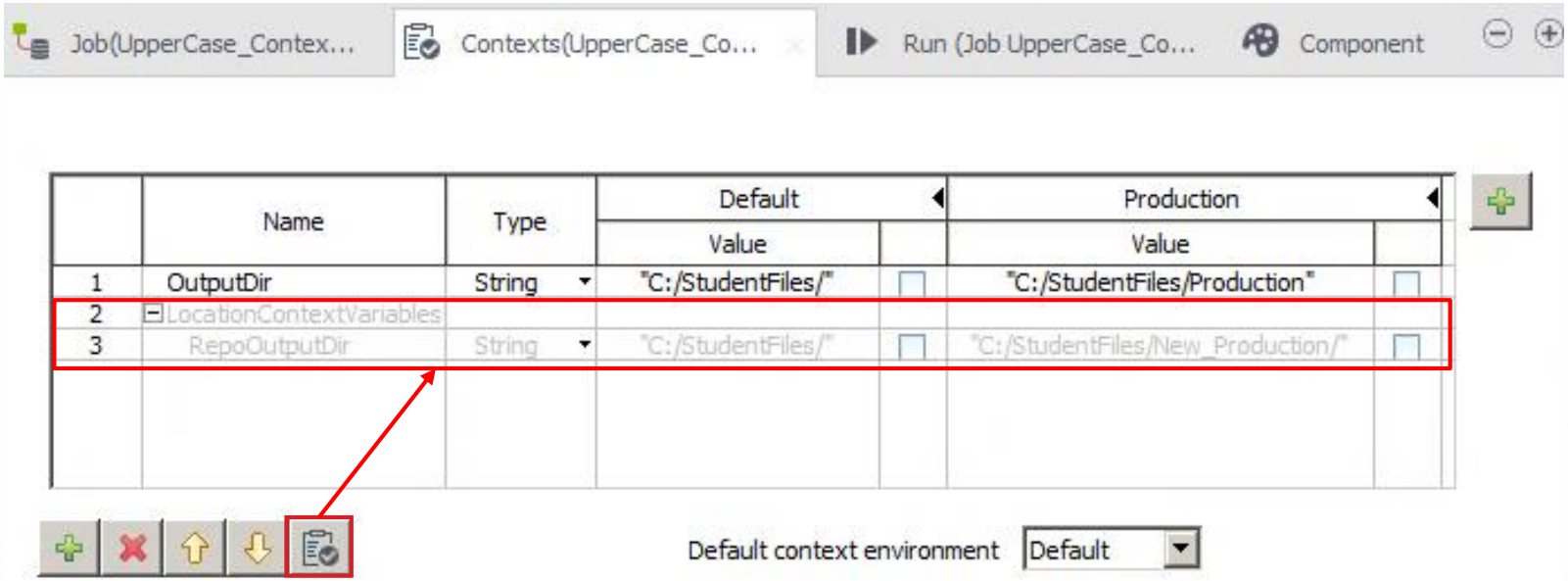
- Add variable(s)
- Add context(s)
- Set default context environment
- **Set pre-defined values**



Update pre-defined values during execution by using the **Prompt checkbox**

Repository Context Variables

- Defined in the **Repository -> Contexts**
  - Similar to Context Variables defined in the Job's **Contexts** tab (Built-In)
- Can be reused in several Jobs
  - Import them in the Job's **Contexts** Tab
  - Values are grayed out - Can only be changed in the **Repository**

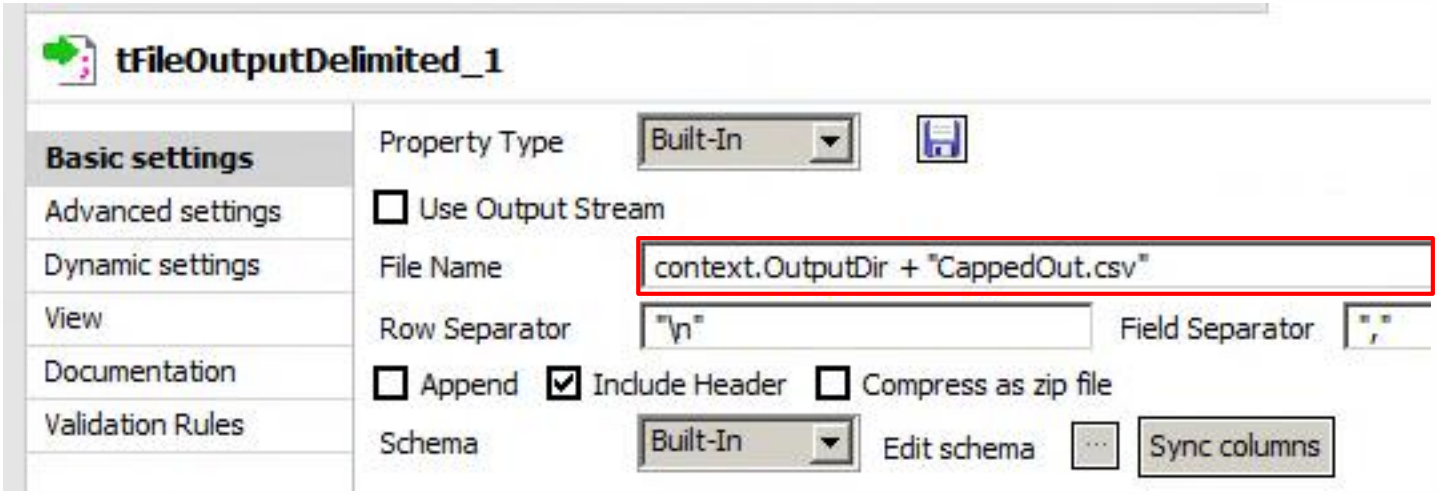


Use Context Variables

Access the Context Variable values

- All Context Variables (Built-In or Repository) are accessed the same way
- Script code to access a context variable:

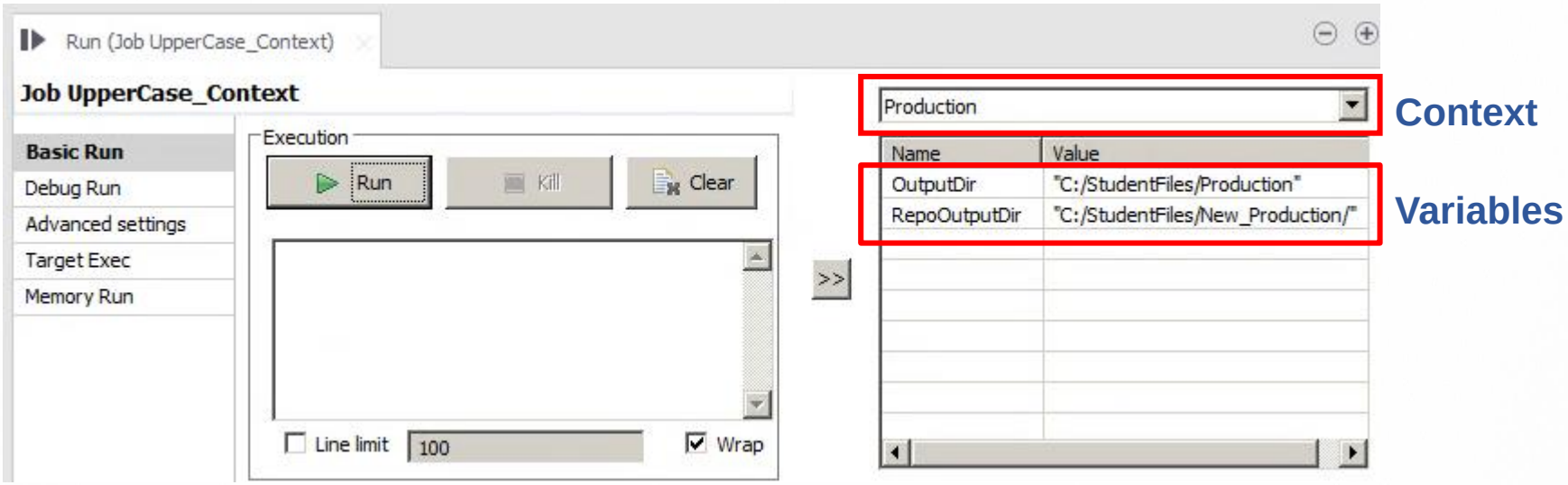
context.<*VariableName*>



Use Context Variables

Change Context While Running a Job

- Specify execution context in the **Run** View
  - The default context and values are displayed
  - When the context is changed, pre-defined values are displayed
    - Pre-defined values can not be updated in the Run View



Lab overview

Create variables to provide different values in two different contexts, **development** and **production** environments:

- Create **context variables**
  - Use the context variable
  - Change context
- Create **Repository context variables**
  - Add repository context variables to a Job
  - Use the repository context variable created
  - Update repository context variable's pre-defined value

Lesson summary

Context Variable types

- **Built-In**
  - Defined for each Job in the **Contexts** View
  - Can be updated in the Job's context view
- **Repository**
  - Defined in the Repository -> Contexts
  - Can be reused in several Jobs
  - Can only be updated in the Repository

Execution Context can be set up in the Run View

Error Handling

Lesson Objectives

By the end of this lesson, you will be able to:

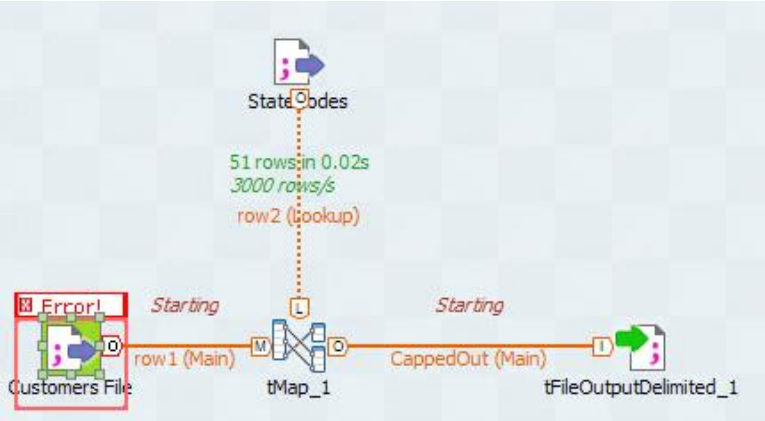
- Perform basic error handling in the Studio
  - Kill a Job when an error occurs
  - Implement specific Job execution path on the component error
- Raise a Warning under specific conditions
- Configure the log level in the run console

Error Handling

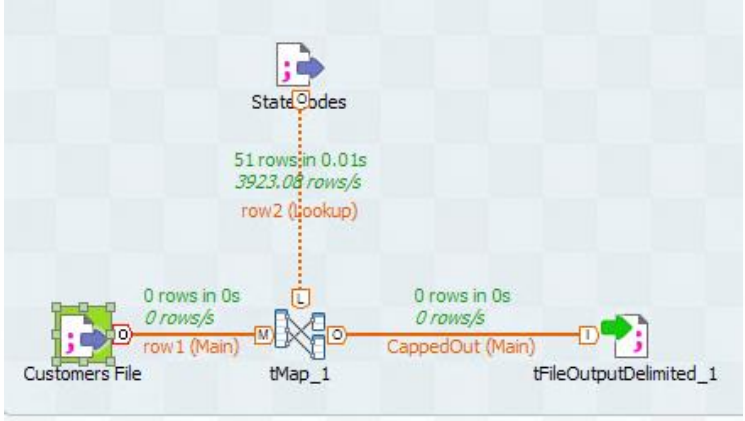
Overview

- In the event that a component throws an error
  - Example: The input file does not exist
- Job execution options:

1. The Job stops when error is encountered



2. The Job completes execution despite the error

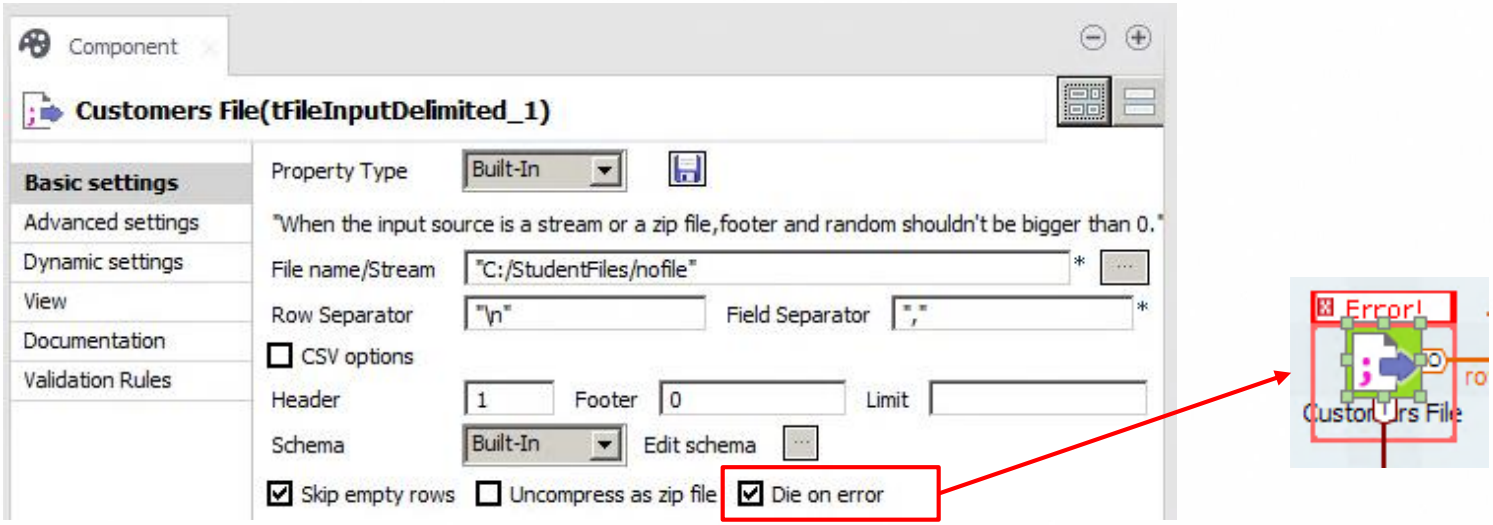


Error Handling

Kill a Job on component Error

- **Die On Error** option - Available for certain components

Check **Die On Error** option → Job stops execution when error is encountered

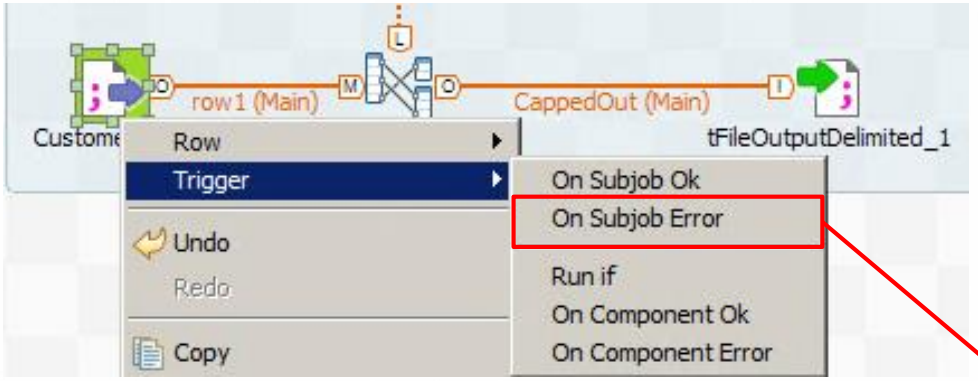


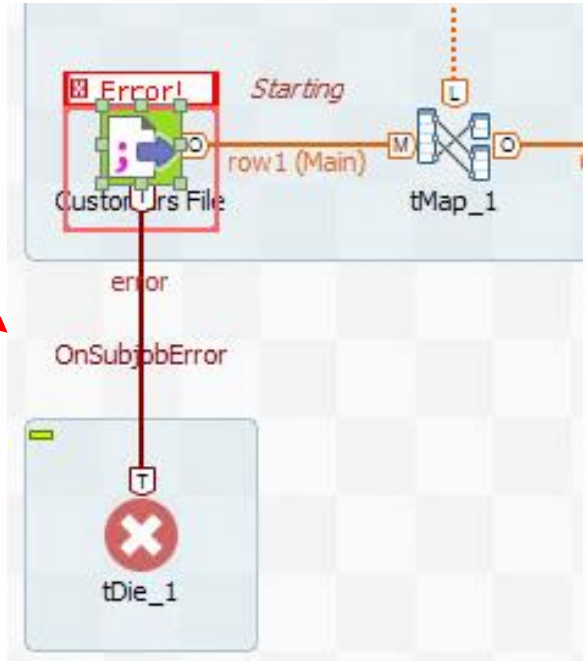
- Some components always die on error, so the option is not available

Error Handling

Triggers

- **Triggers** allow implementation of different execution paths:
  - The status of components/subjobs
  - Specific conditions





- Example: Define a message in case of an error
  - Use **tDie** component to **set the Die Message**

Error Handling

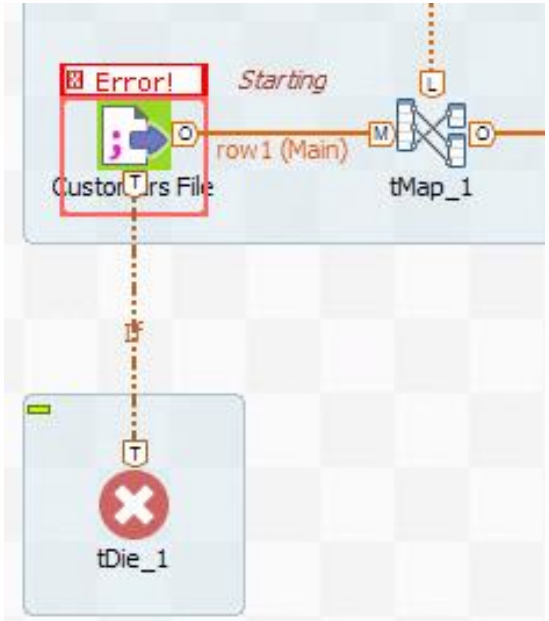
Kill a Job under specific conditions

Requirement:

- Kill a Job under specific conditions (no technical error)
- Ex: The input file has less then a certain number of lines

Solution:

- Use the **tDie** component to **kill the Job**
- Use **Run If** trigger to **specify the condition**
  - The Job execution stops if the condition is met



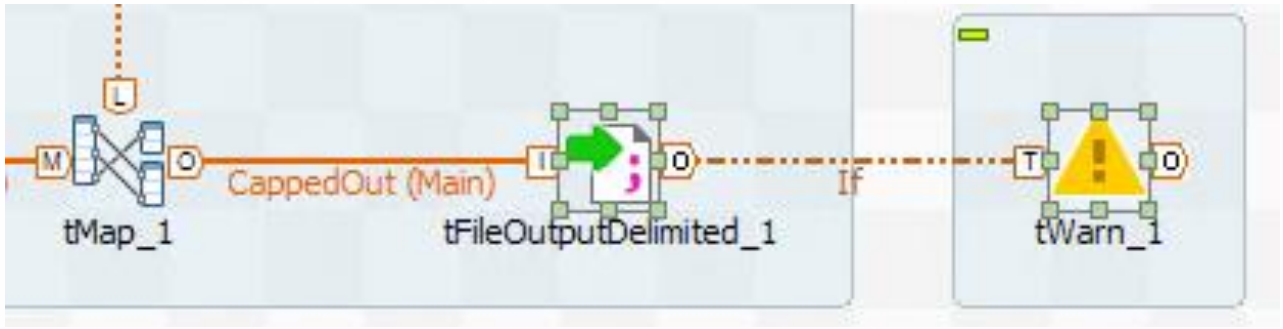
Raise a Warning

Need:

- Raise a Warning when a business rule is not met
- Example: The output file does not have the expected number of lines

Solution:

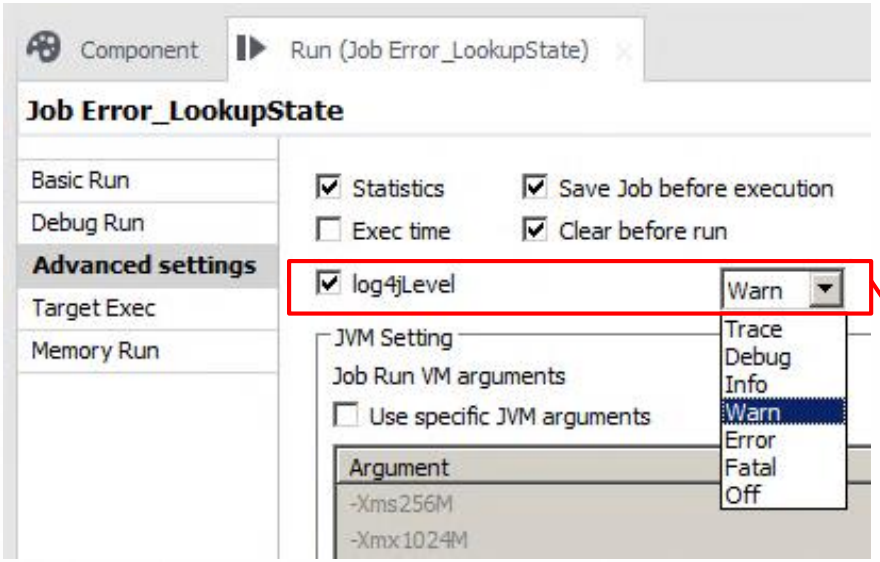
- Use the **tWarn** component to **raise a warning**
- Use **Run If** trigger to **specify the condition**
  - A warning is raised if the condition is met

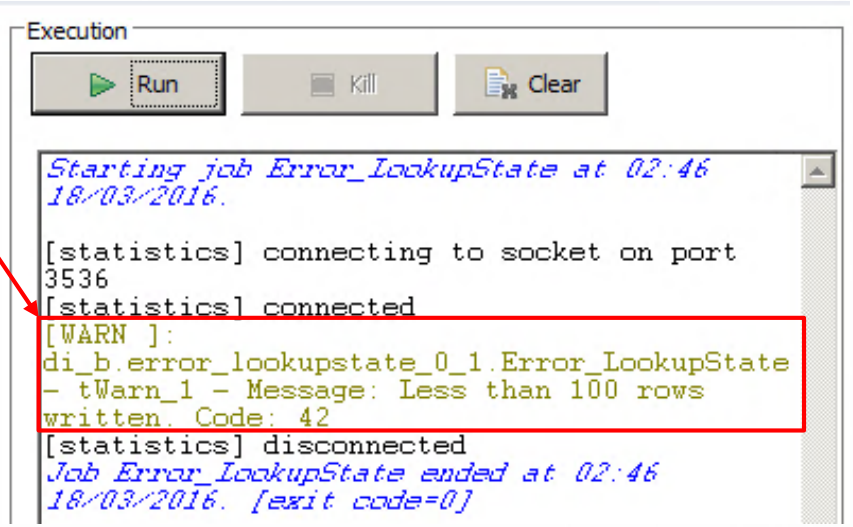


Log Level

Configure the log level

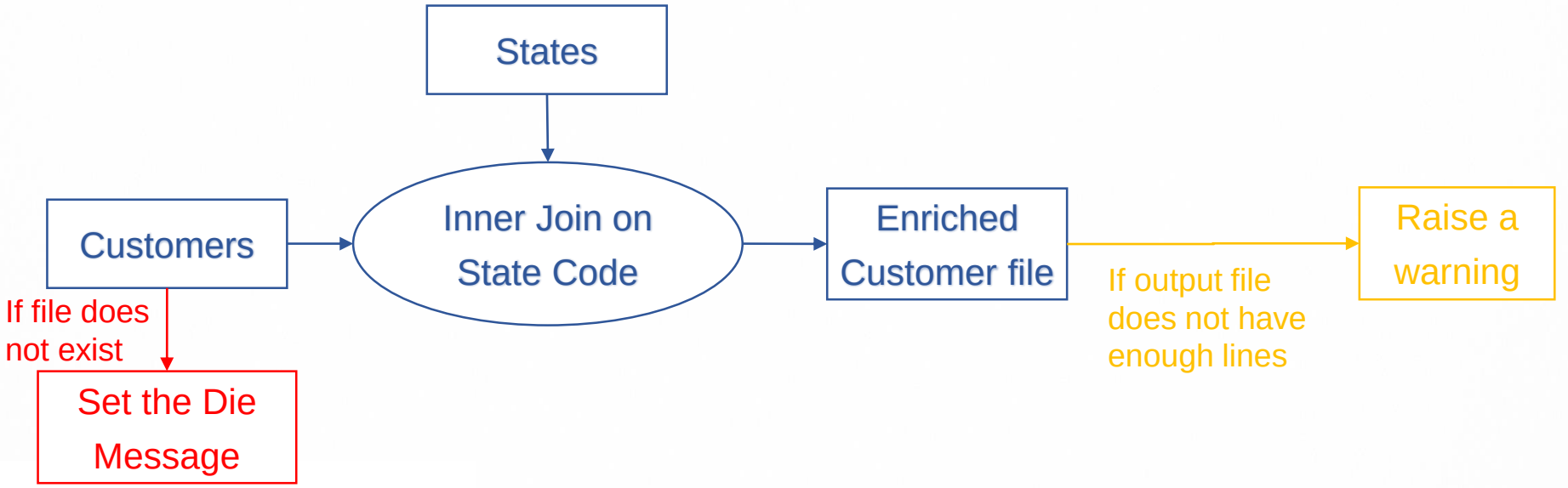
- **log4jLevel**
  - Available under the **Advanced Settings tab** of the **Run** View
  - Specifies the types of **log** messages to be displayed





Lab overview

1. Create a technical error: input file does not exist
   - Kill the Job
   - Set the Die message
2. Create a business error: output file does not have enough lines
   - Raise a warning



Lesson summary

- Handling Errors
  - **Die On Error** option – Kills a Job on component error
  - **tDie** component can be used:
    - To set the Die Message
    - To kill a Job under certain conditions
- Raise a Warning
  - **tWarn** component can be used
- Configure **log level** in the Advanced setting of the **Run** view

Generic Schemas

Lesson Objectives

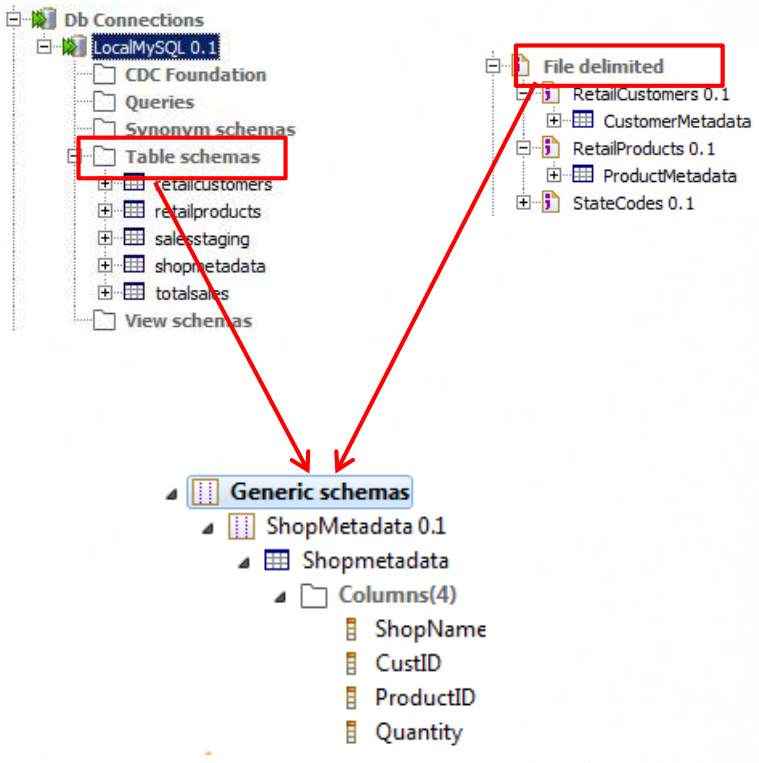
By the end of this lesson, you will be able to:

- Create a Generic Schema
- Generate files containing random data
- Capture all information on your files in context variables
- Use subjobs

Generic schemas - Definition

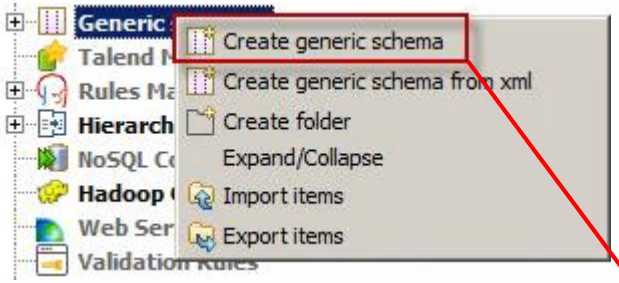
It is possible to create schema metadata for specific types of Files (Delimited, XML, JSON, etc). But this schema metadata can't be used interchangeably as they are linked to a type of file.

A **generic schema** is one that can be used in any component requiring a schema and is **not linked to any type of source** (file, database, etc.).

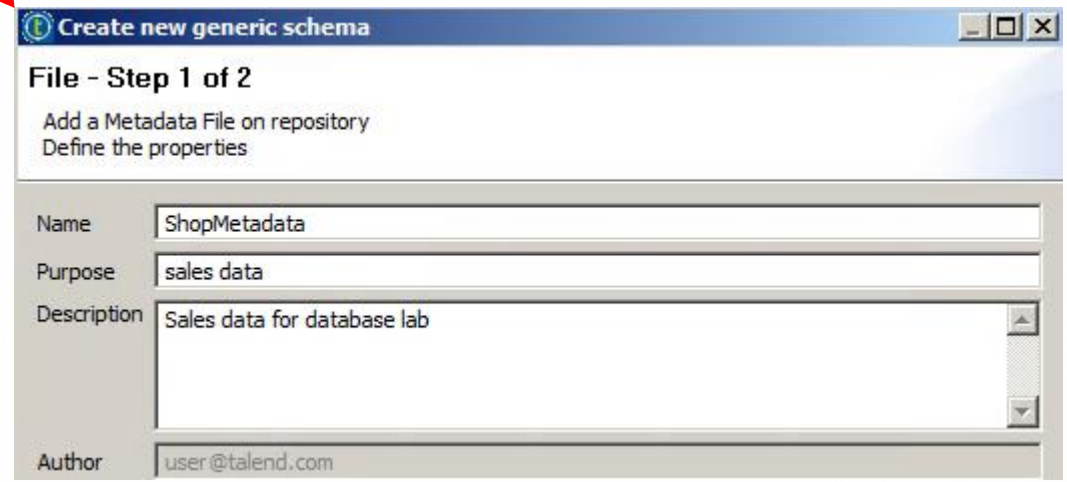


Create New Generic Schema

- **Generic Schemas** can be found under **Repository > Metadata**

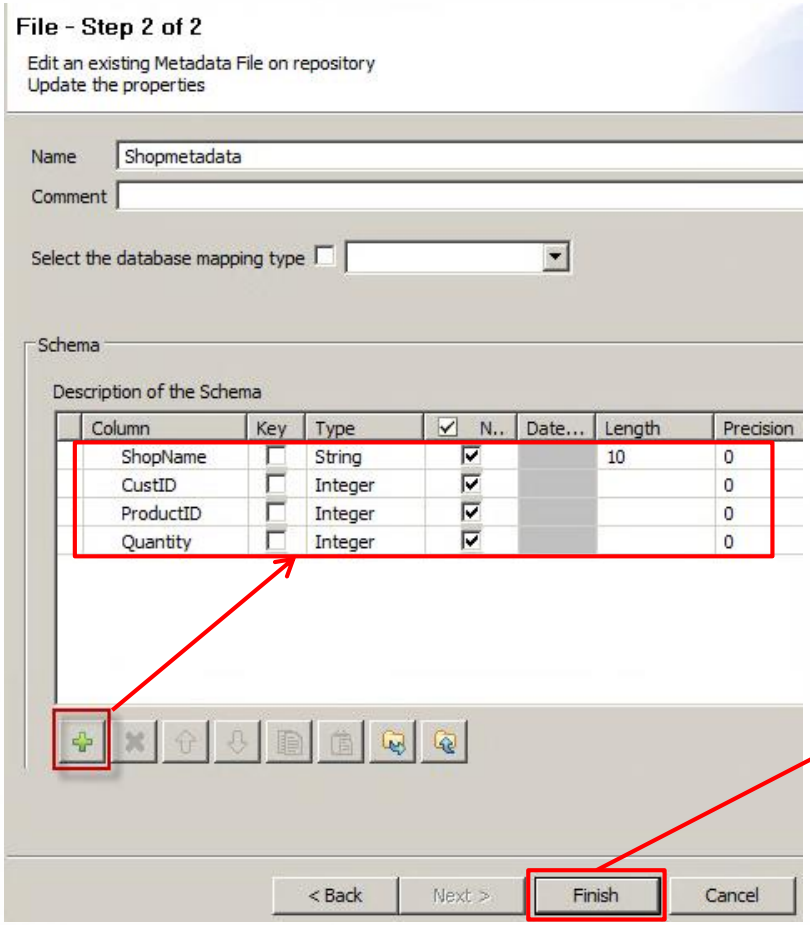


Provide **Name**, **Purpose** and **Description**.

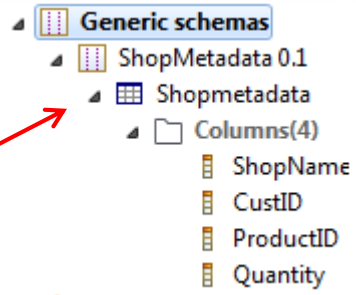


Create New Generic Schema

- Specify the Column's **Name** and **Type**

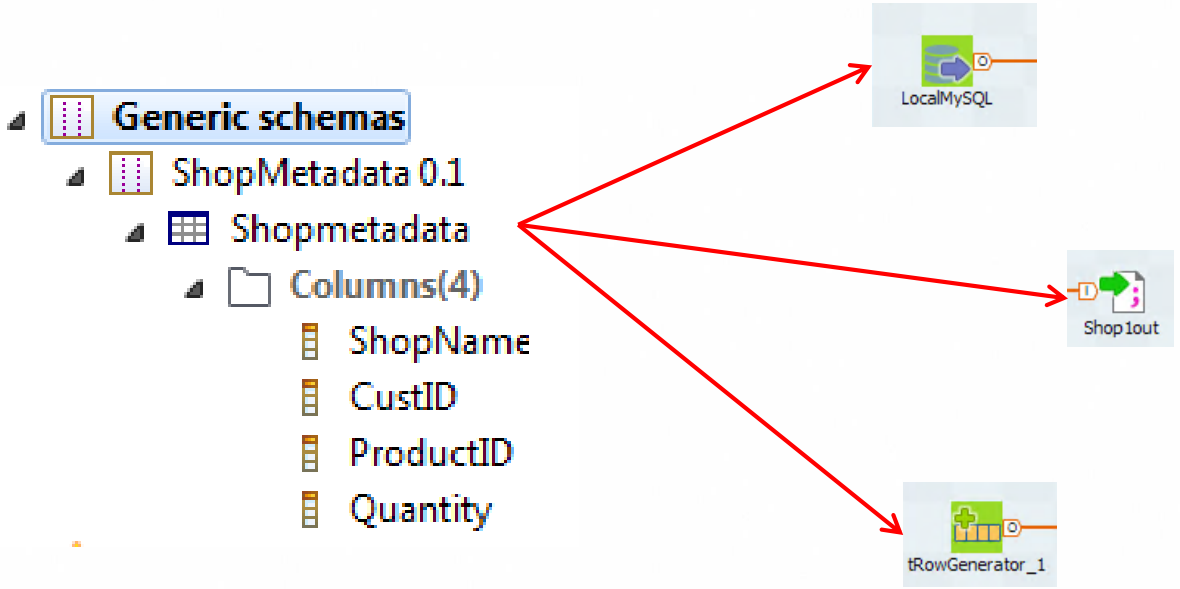


The Generic schema appears in the Repository



Where are Generic schemas used?

- Generic schemas can be used in various components:
  - Input components: **tFileInputDelimited**, **tMySqlInput**…
  - Output components: **tFileOutputDelimited**, **tMySqlOutput**…
  - Miscellaneous components: **tRowGenerator**…



Generating Random Data

Using the tRowGenerator component

Define schema

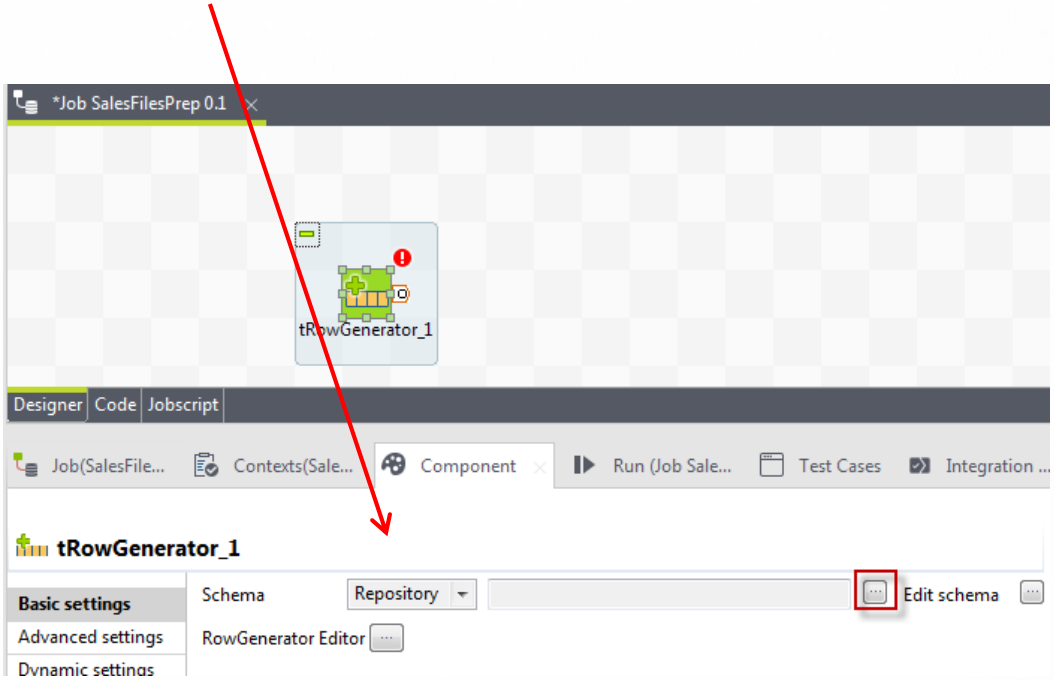
Set number of rows

Select Functions

Set environment variable values

Preview values

- The first step is to specify the schema:
  - Built-In
  - Repository



Generating Random Data

Using the tRowGenerator component

Define schema

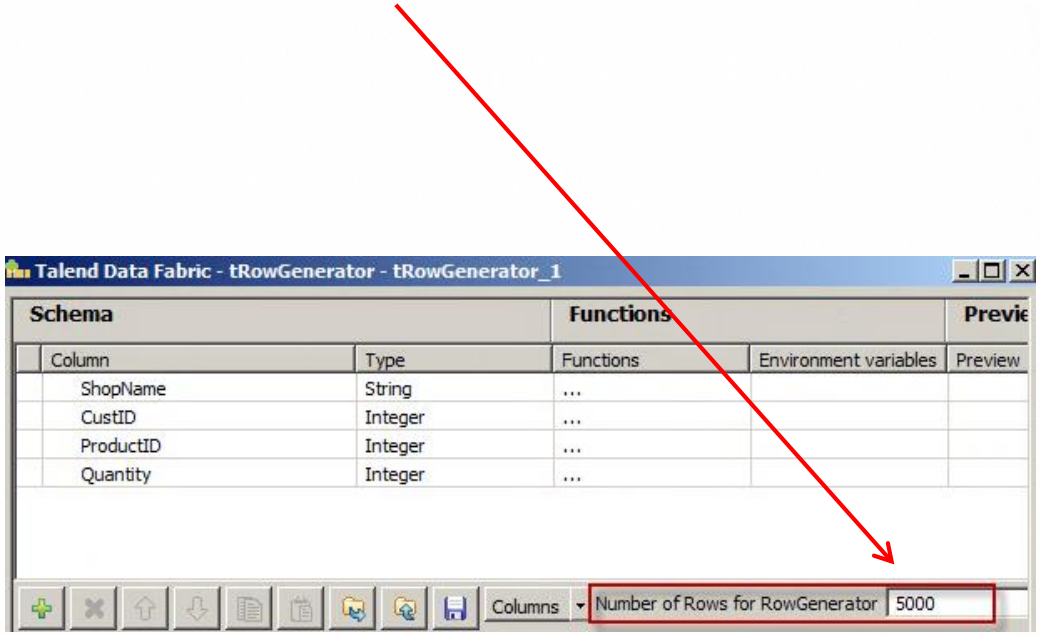
Set number of rows

Select Functions

Set environment variable values

Preview values

- Specify the number of rows to generate in the **tRowGenerator** editor



Generating Random Data

Using the tRowGenerator component

Define schema

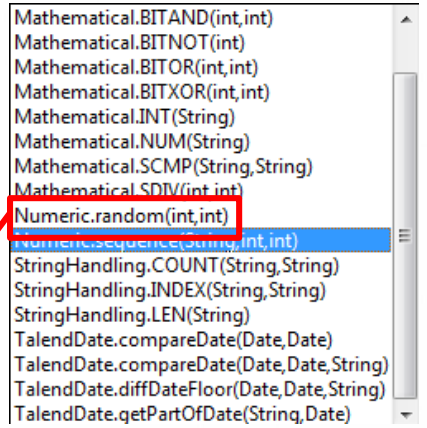
Set number of rows

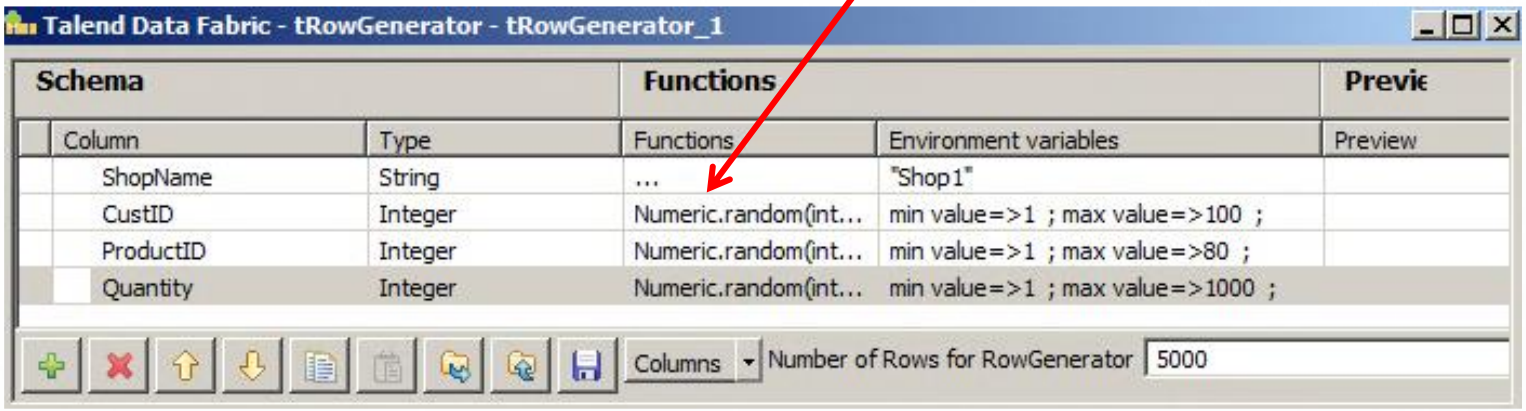
Select Functions

Set environment variable values

Preview values

- For each column in the schema, select the **Function** that will be used to generate the data.





Generating Random Data

Using the tRowGenerator component

Define schema

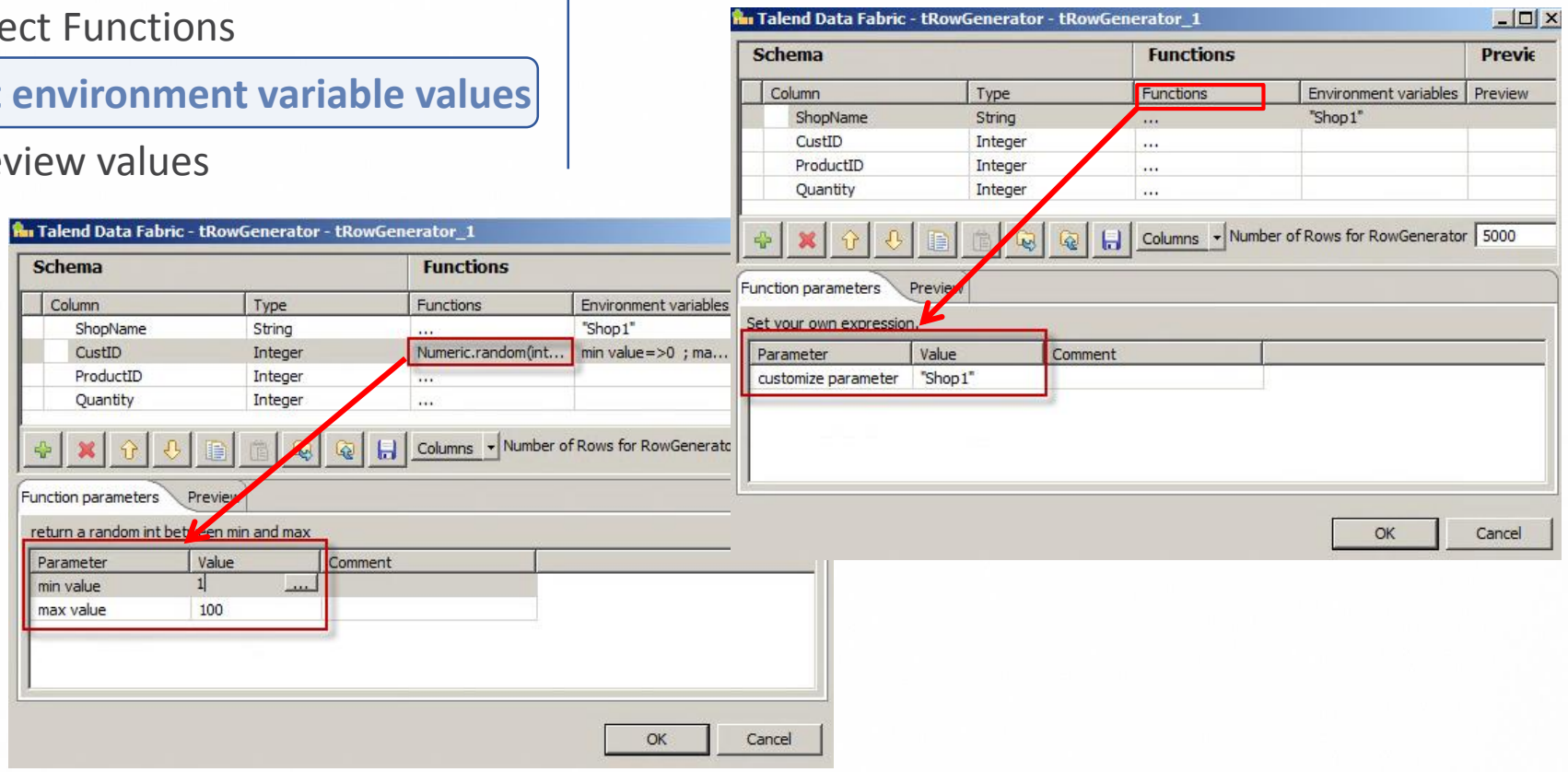
Set number of rows

Select Functions

Set environment variable values

Preview values

- Depending on the selected Function, set the corresponding **parameters**.



Generating Random Data

Using the tRowGenerator component

Define schema

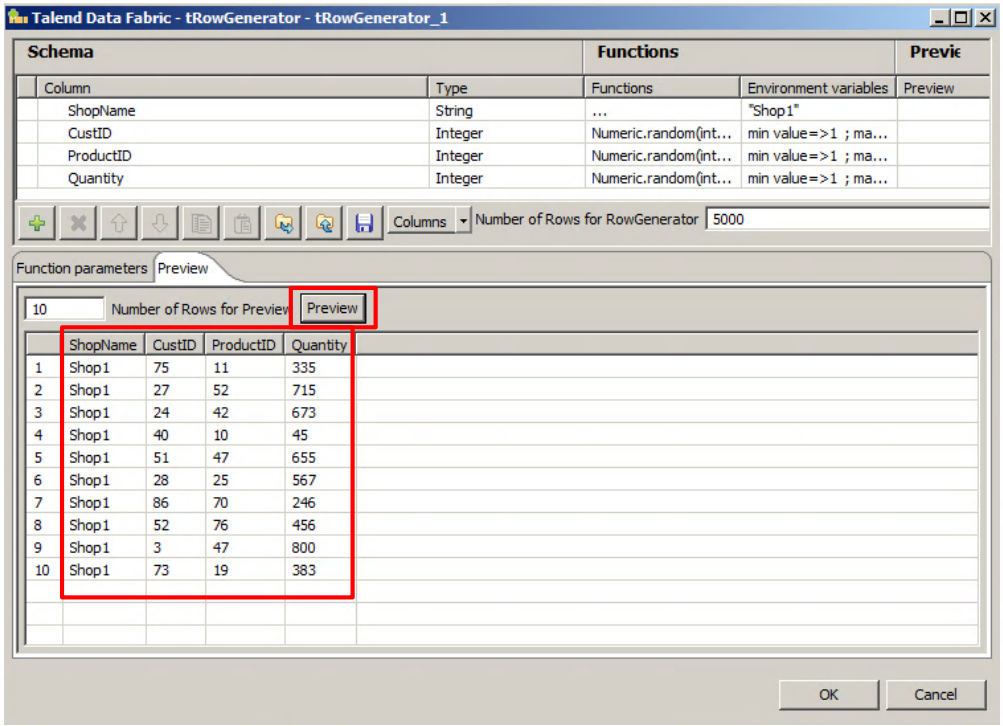
Set number of rows

Select Functions

Set environment variable values

Preview values

- Verify the configuration using the **Preview** feature of the **tRowGenerator** component



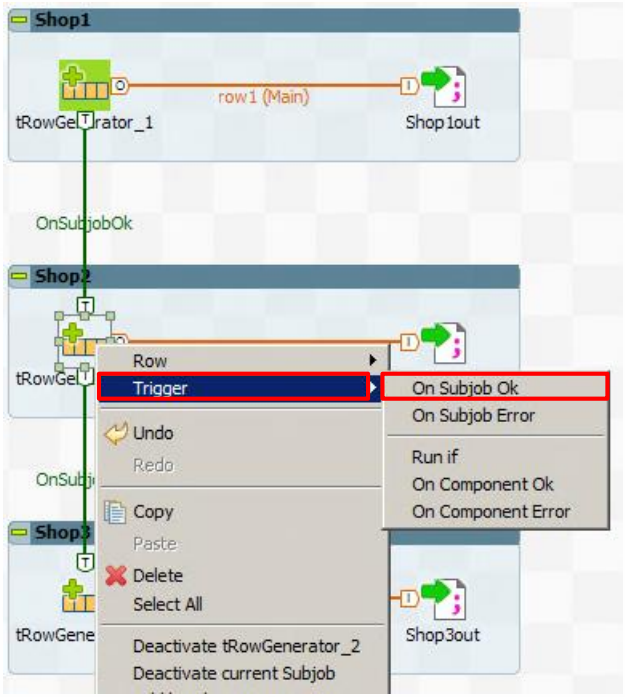
What is a Subjob ?

- A collection of connected components within a Job

Each subjob appears in a blue rectangle

Connecting Subjobs with Triggers

- A subjob is connected to another subjob with a **Trigger**, available on the *first* component of the Subjob.



The green links (Triggers) are **not flows**

- No data is transferred from one component to another
- They make it possible to **sequence** the execution of Job Design processing

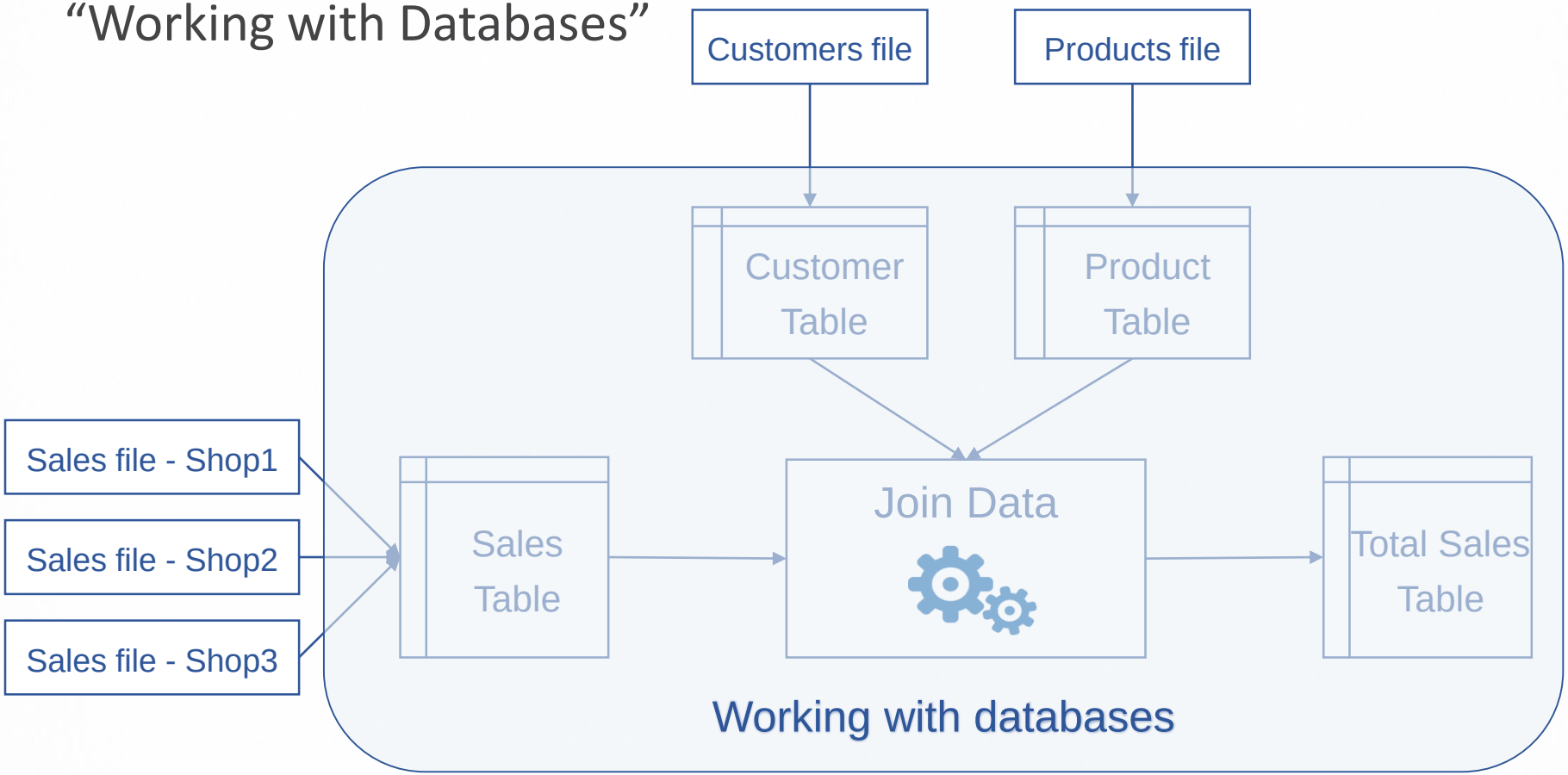
Subjobs – Activation/Deactivation

- Any subjob (or component) can be deactivated/activated.
- Once deactivated, the subjob components appear grayed-out.

Lab overview

Generic Schemas

- Prepare schemas, files, and metadata for the next lesson "Working with Databases"



Lesson summary

- **Generic Schemas** are reusable schemas not linked to any type of source
- **tRowGenerator** paired with Generic Schemas can be used to produce sample data
  - Great for testing
  - Easy to test edge case values (e.g. boundaries such as low/high limits)
- **Subjobs with Triggers** - Process data based on different flows
  - When things work properly (On Subjob OK)
  - When things don’t (On Subjob Error)

Working With Databases

Objectives

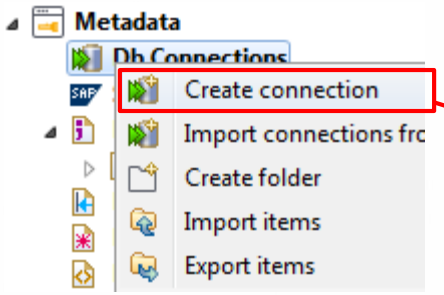
By the end of this lesson, you will be able to:

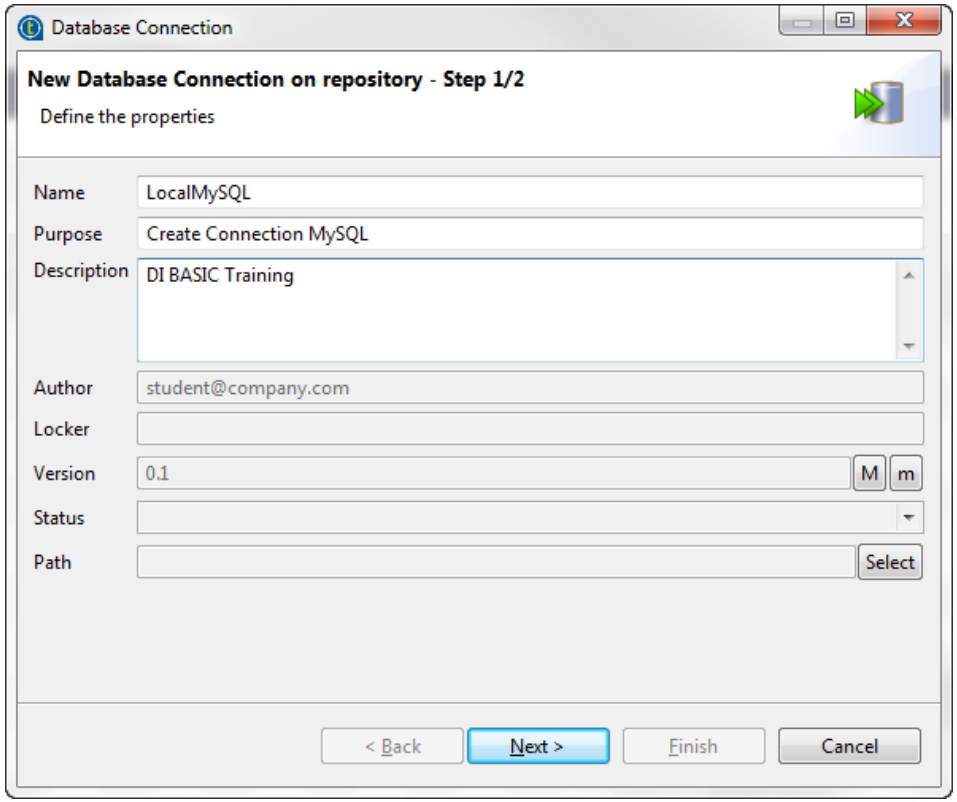
- Connect to a database from a Talend Job
- Use a component to create a database table
- Write to and read from a database table
- Filter unique data rows
- Perform aggregation
- Write data to an XML file

Database Connection

Creating Metadata

- Database connection metadata can be found under **Repository > Metadata > Db connections**



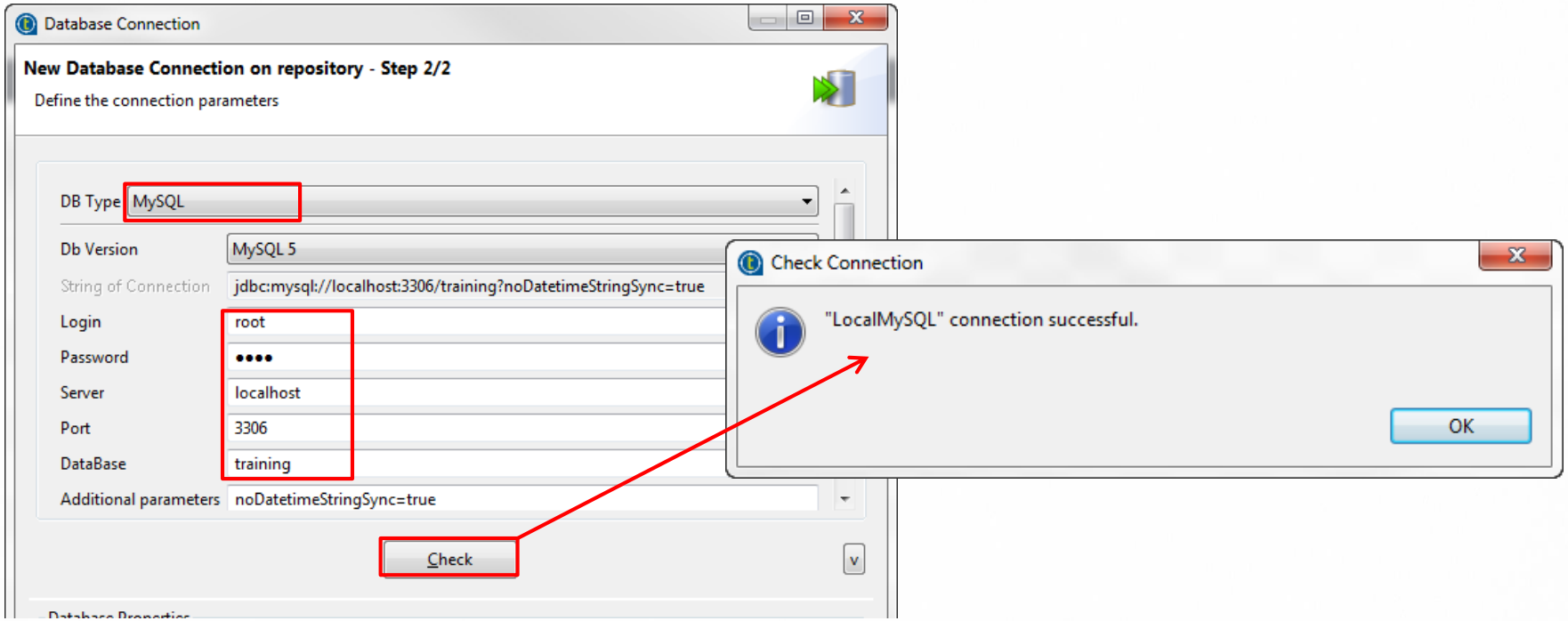


Provide **Name**, **Purpose**, and **Description**

Database Connection

Creating Metadata

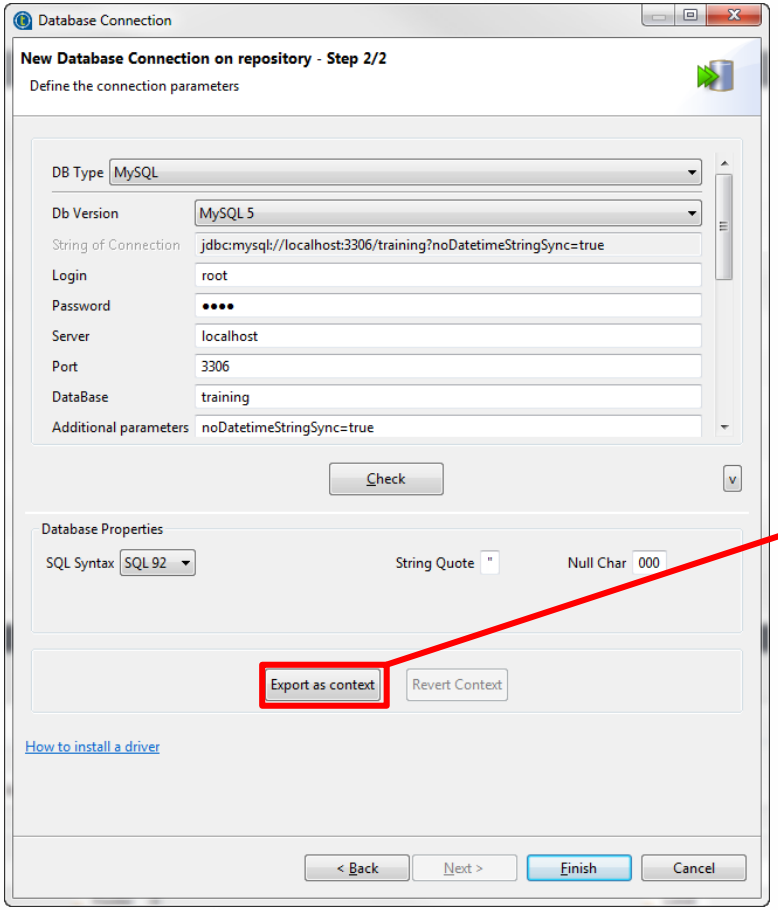
- Specify **key connection information** and **check** your connection

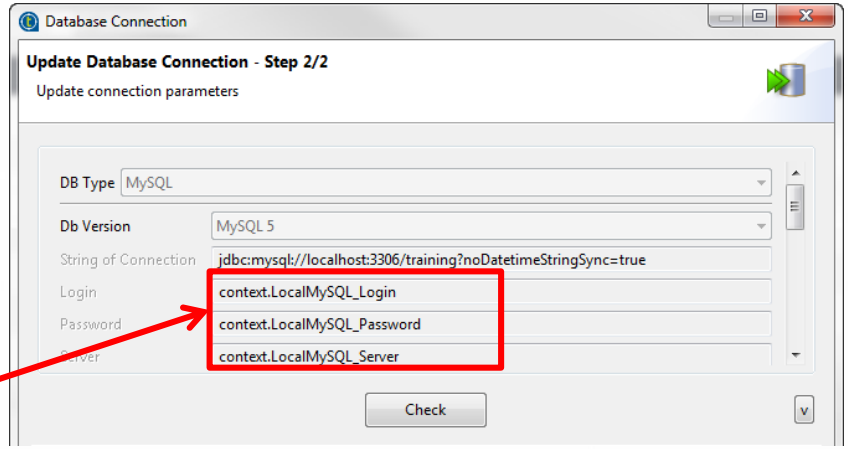


Database Connection

Export as Context

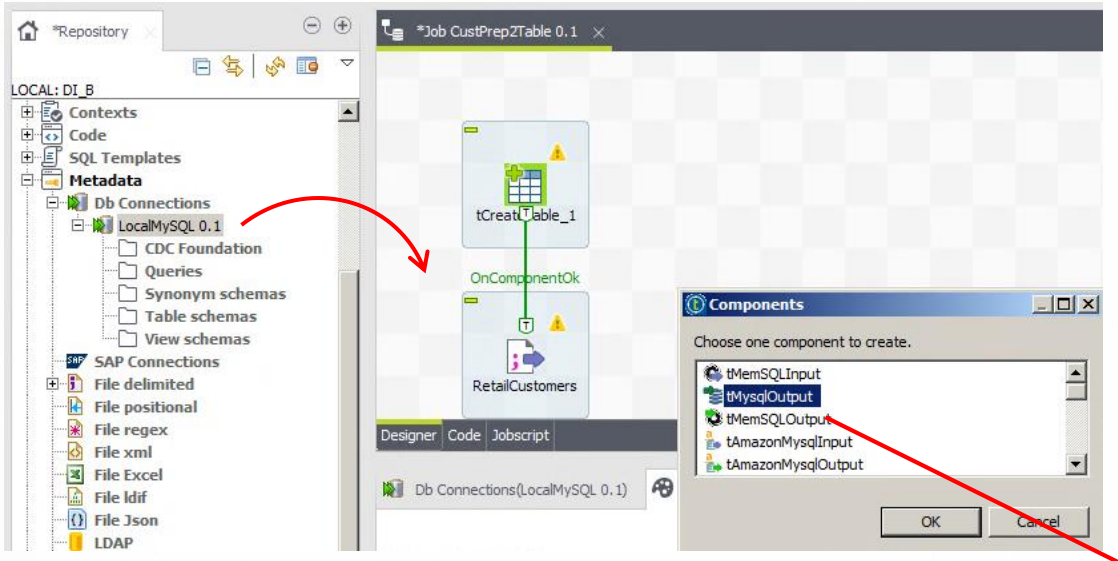
- Save connection info as context variables



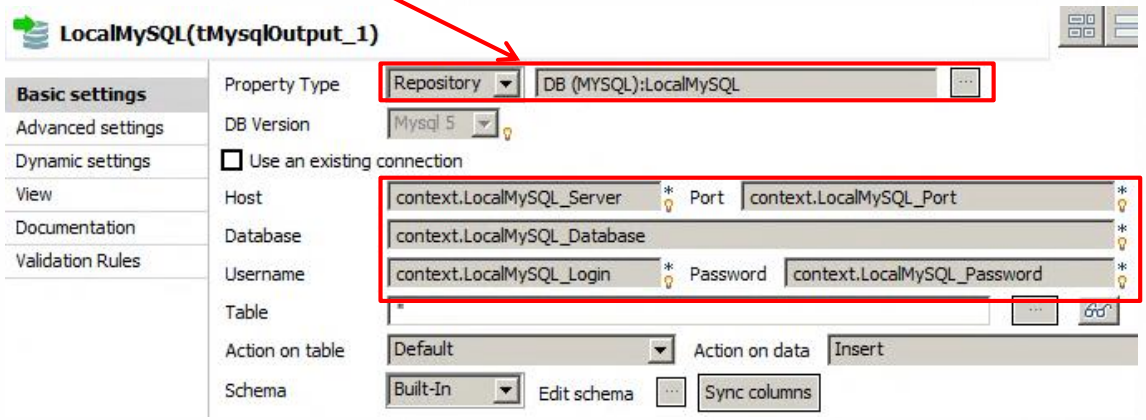


Database Connection

Using Metadata



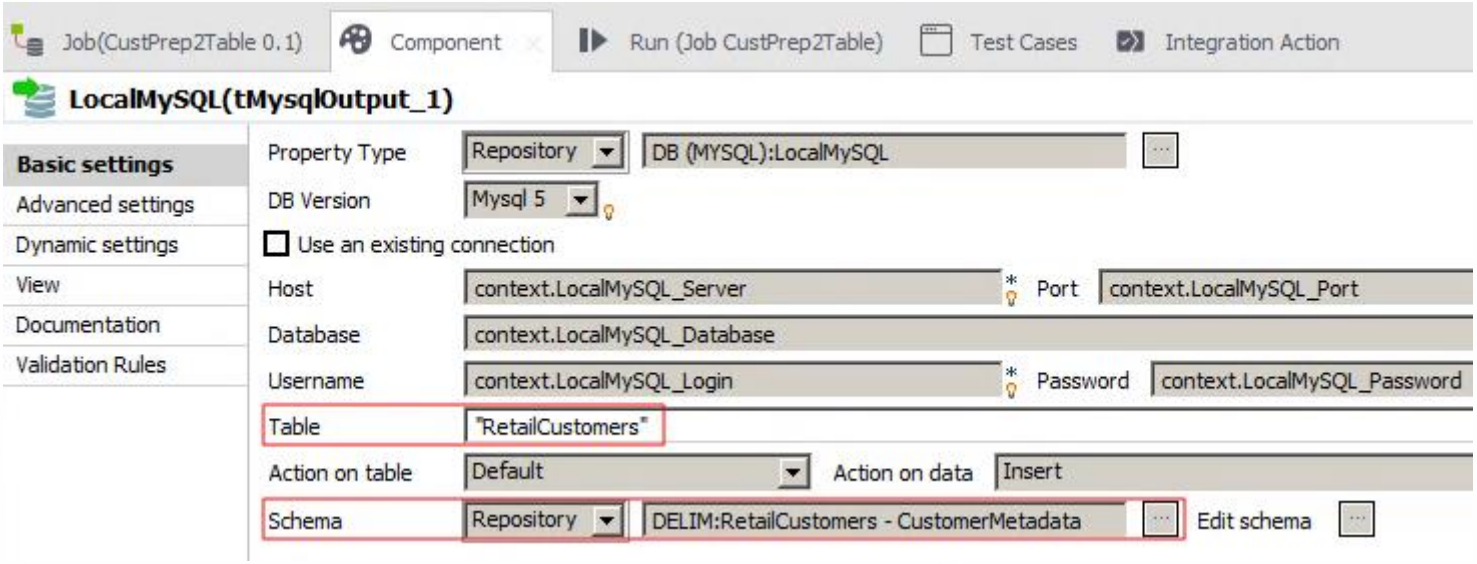
- Drag the database connection metadata to the Designer
- Select a component in the list
- The component is configured with the **connection information metadata**



Writing data to a Database

tMysqlOutput

- To write data to a MySQL database, you can use the **tMysqlOutput** component

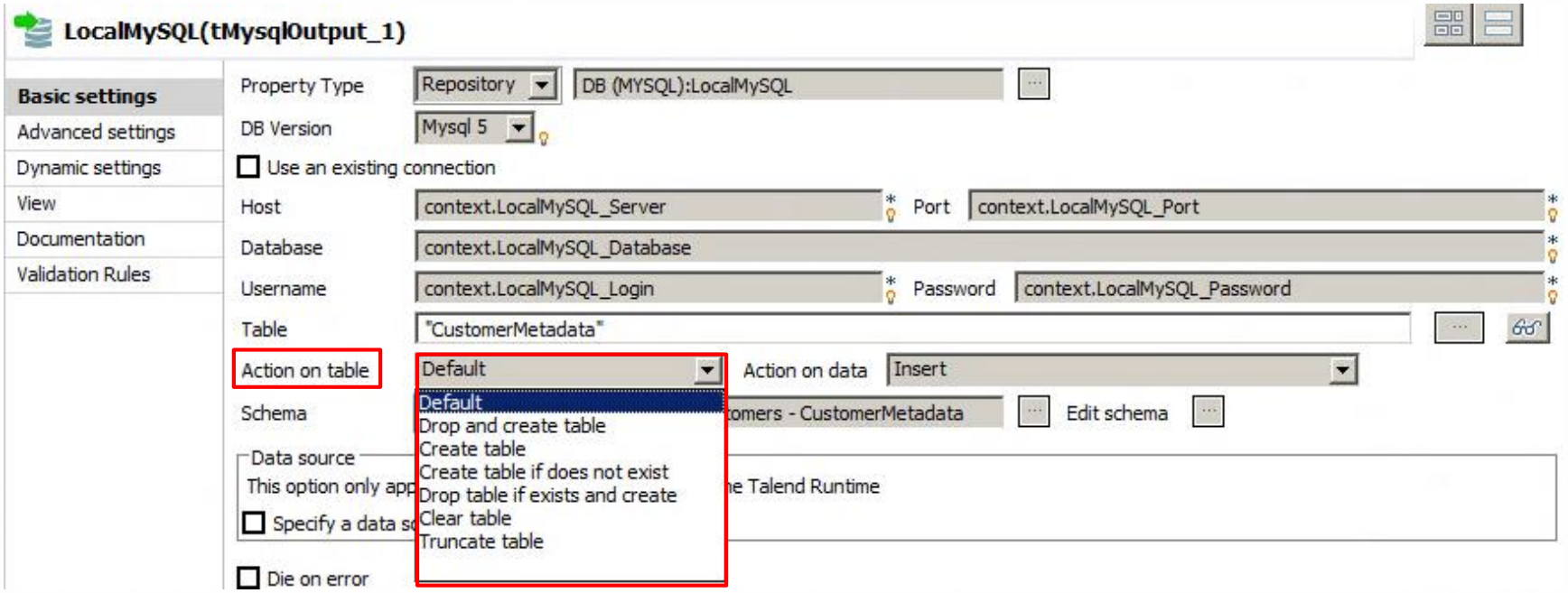


- You should also provide the **Table Name** and the **Schema**

Writing data to a Database

tMysqlOutput – Action on table

- Select the **Action on table** in the drop down list

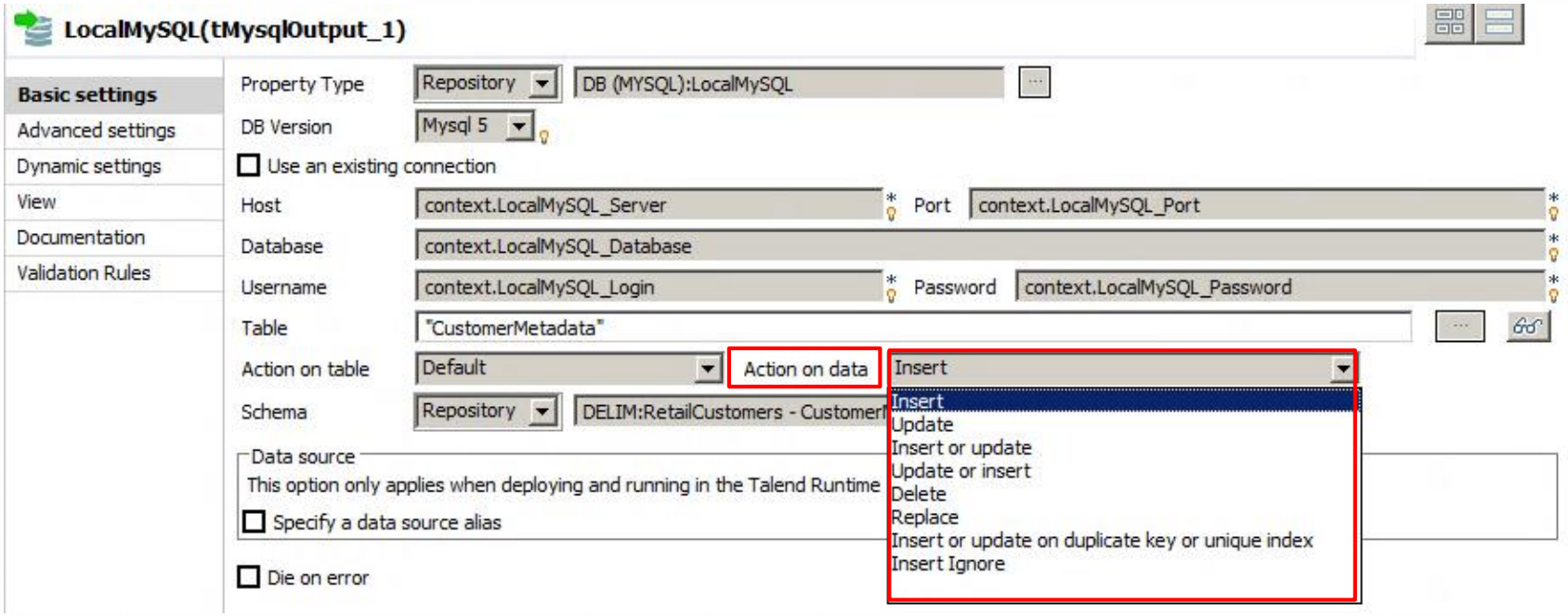


- The table can be created first (if needed) then written to

Writing data to a Database

tMysqlOutput – Action on data

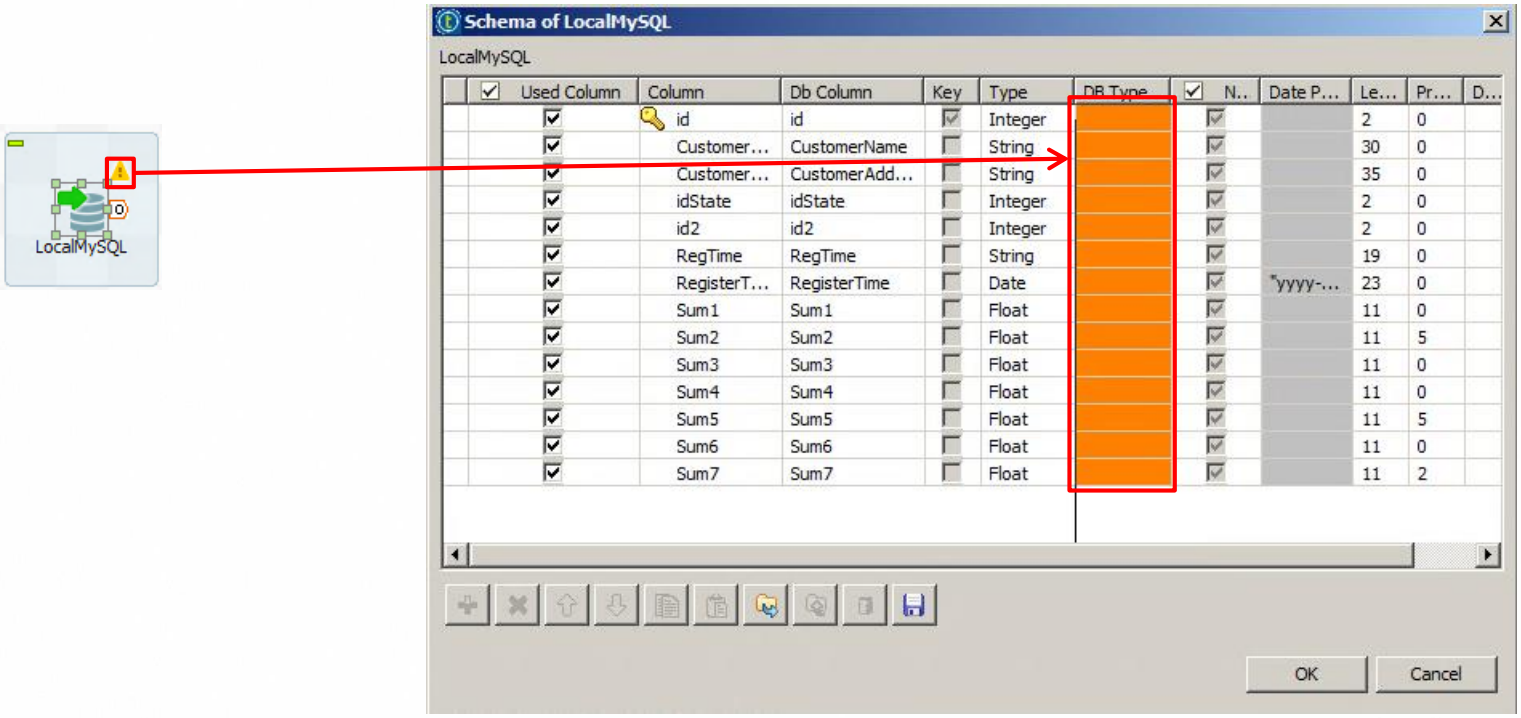
- Select the **Action on data** in the drop down list



Writing data to a Database

tMysqlOutput – Fix schema

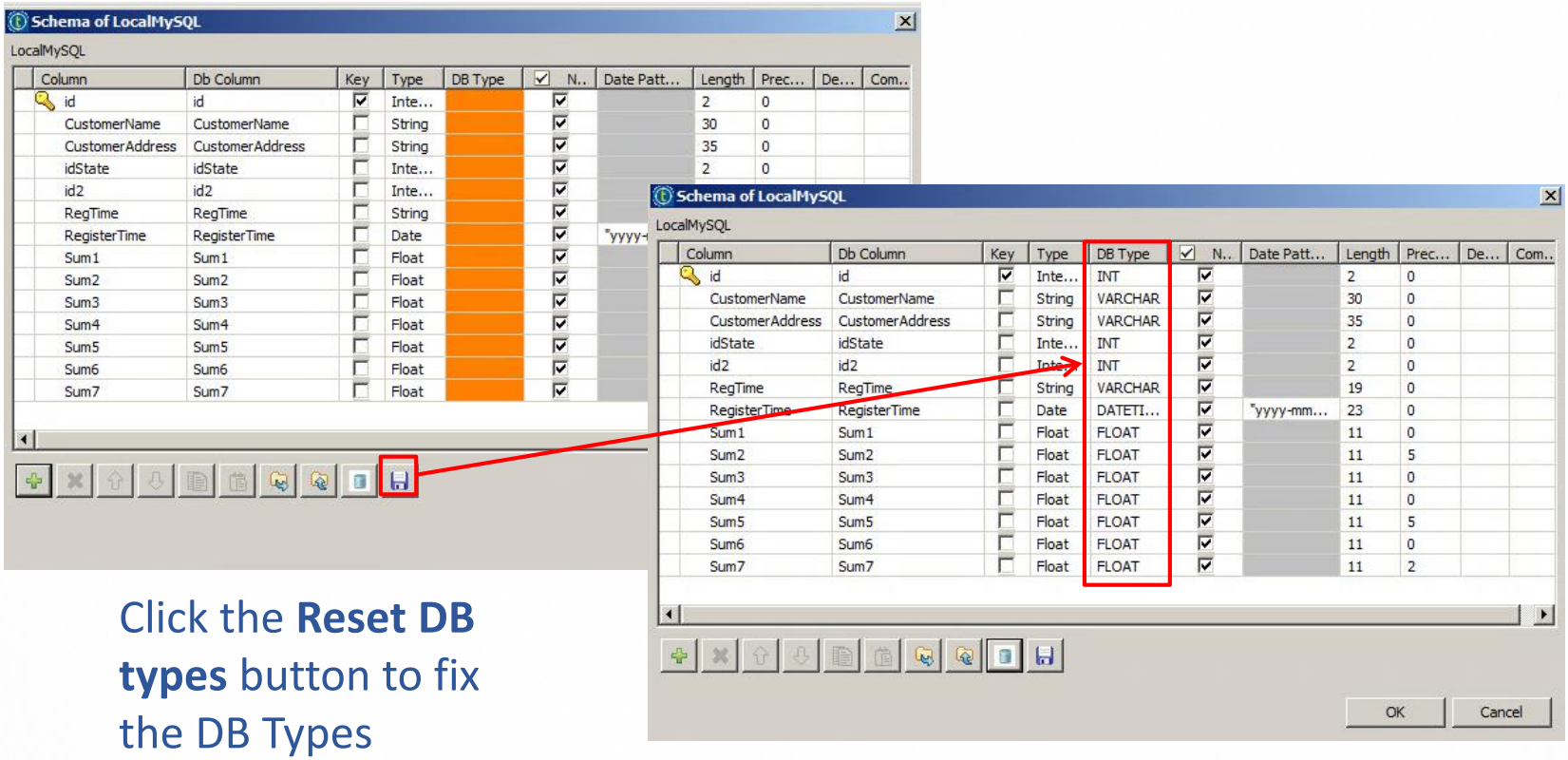
- The schema in a **tMysqlOutput** component requires Database column names and types



Writing data to a Database

tMysqlOutput – Fix schema

- The schema in a **tMysqlOutput** component requires Database column names and types



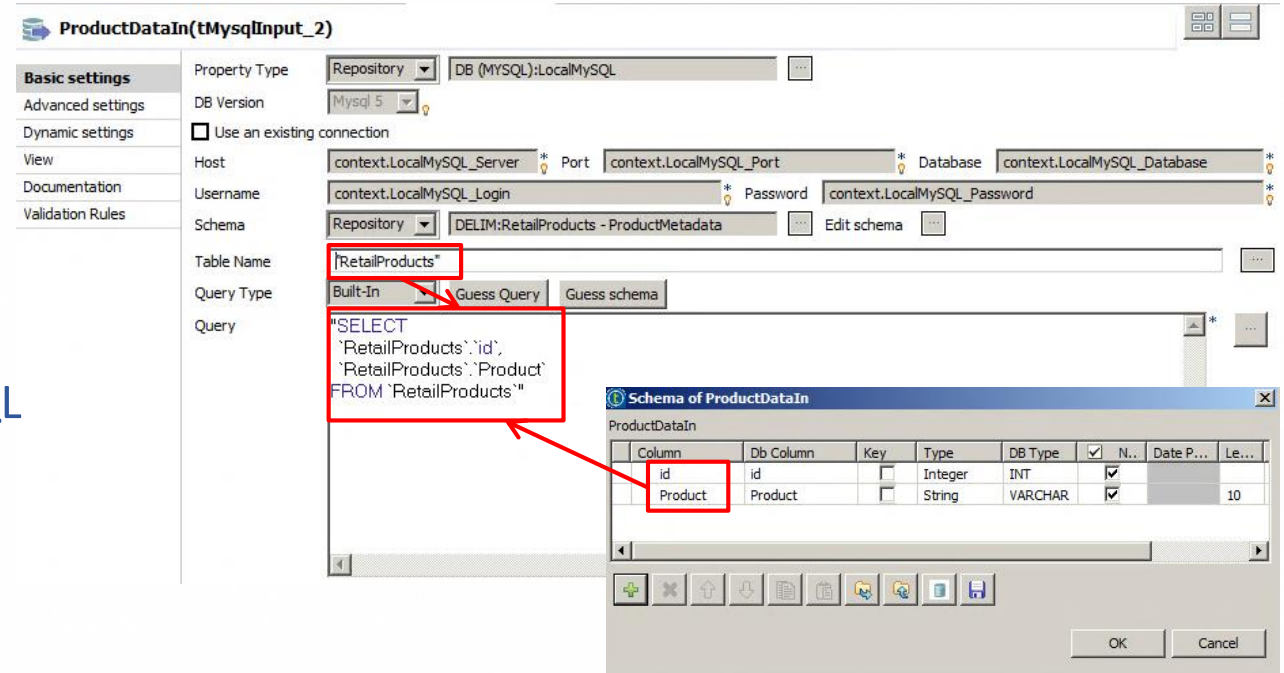
Click the **Reset DB types** button to fix the DB Types

Reading from a Database

tMysqlInput

To read data from a MySQL database, you can use the **tMysqlInput** component. This component requires a **SQL Query** to read the data from the MySQL database.

Click the **Guess Query** button to build the SQL Query based on the **Table Name** and the **Schema**



Processing Data

Other components that are commonly used to help process data include:

Name: tSortRow

Purpose:

- Sort input data based on one or more columns
- Specify sort type and order

Name: tUniqRow

Purpose: Filters out duplicate entries

Name: tAggregateRow

Purpose:

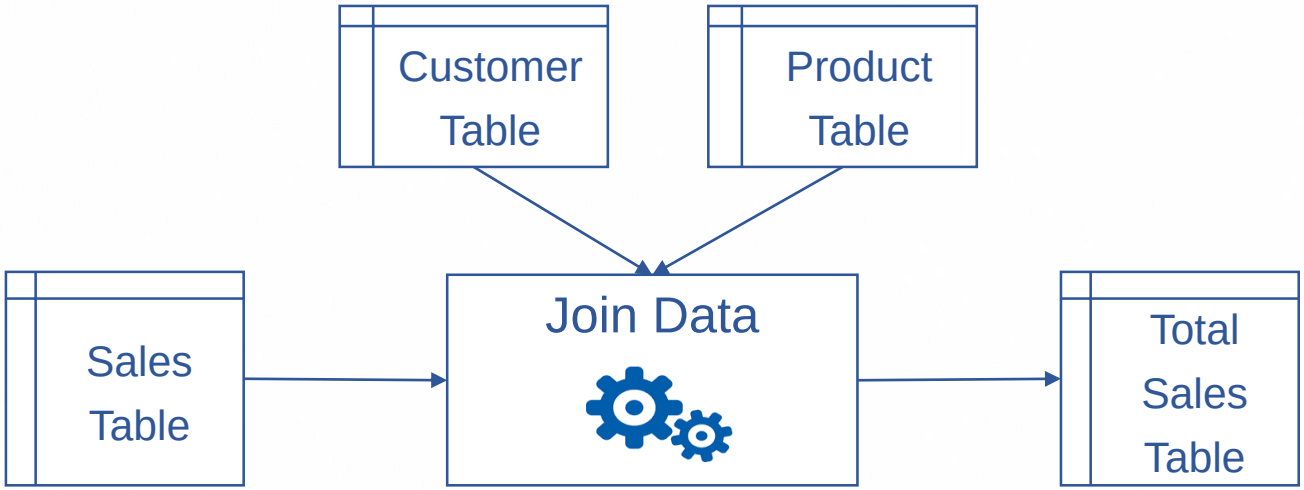
- Aggregates input data based on one or more columns
- Sample operations: min, max, avg, sum, first, last, ….

Lab overview

Working with Databases

- First, you will create Customer, Product, and Sales tables.

- Then, you will join the Sales table to the Customer and Product Tables to generate the Total Sales Table.



Lesson summary

- Create database connection information as reusable metadata
- Read/Write to Databases
  - tCreateTable
  - tMysqlInput
  - tMysqlOutput
- Processing data with components
  - tSortRow
  - tAggregateRow
  - tUniqRow

Master Jobs

Objectives

By the end of this lesson, you will be able to:

- Create a Master Job
- Override context variables
- Export a Job and its dependencies

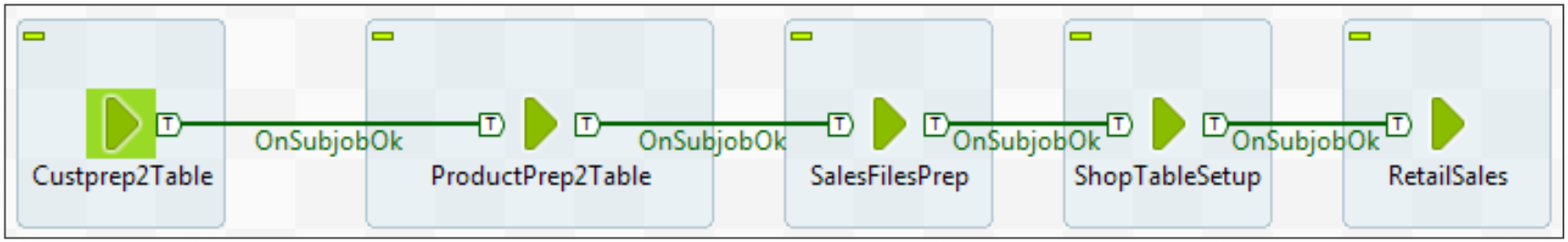
Job Orchestration - Master Jobs

Master Job – Calls and orchestrates child jobs

Job Orchestration - Master Jobs

When, Why and How

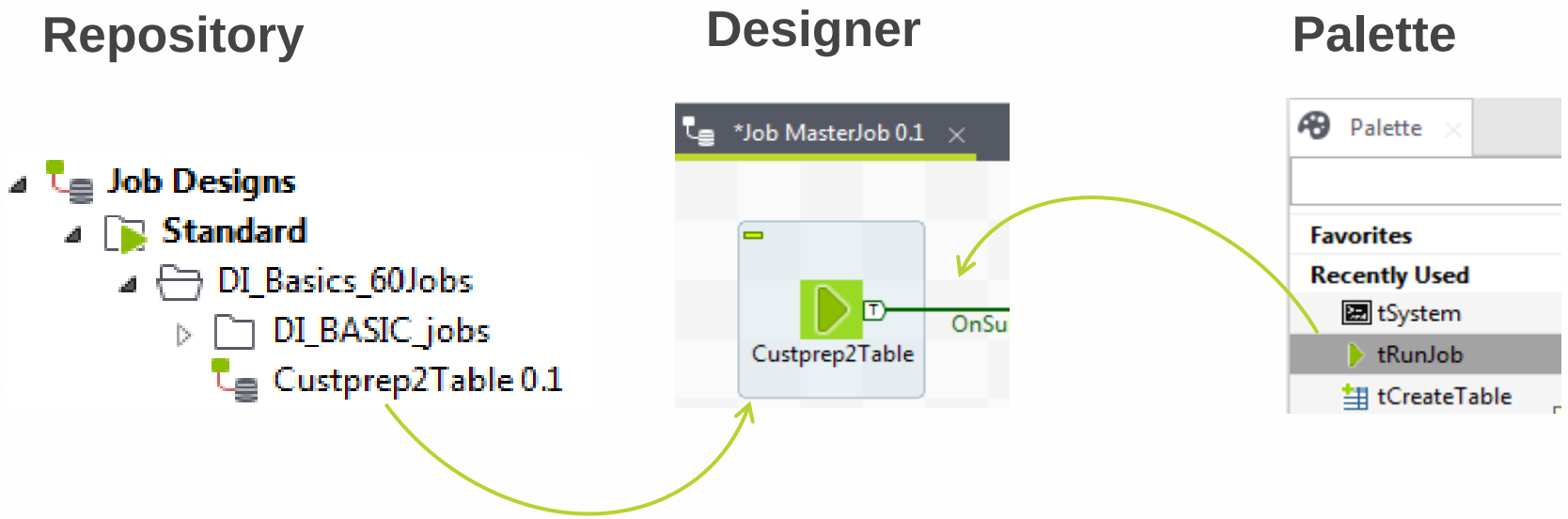
- When?
  - Complex Jobs that need to run in sequence
- Why?
  - Easy to use orchestration layer within the Talend Studio
- How?
  - Largely through **tRunJob** component



Job Orchestration - Master Jobs

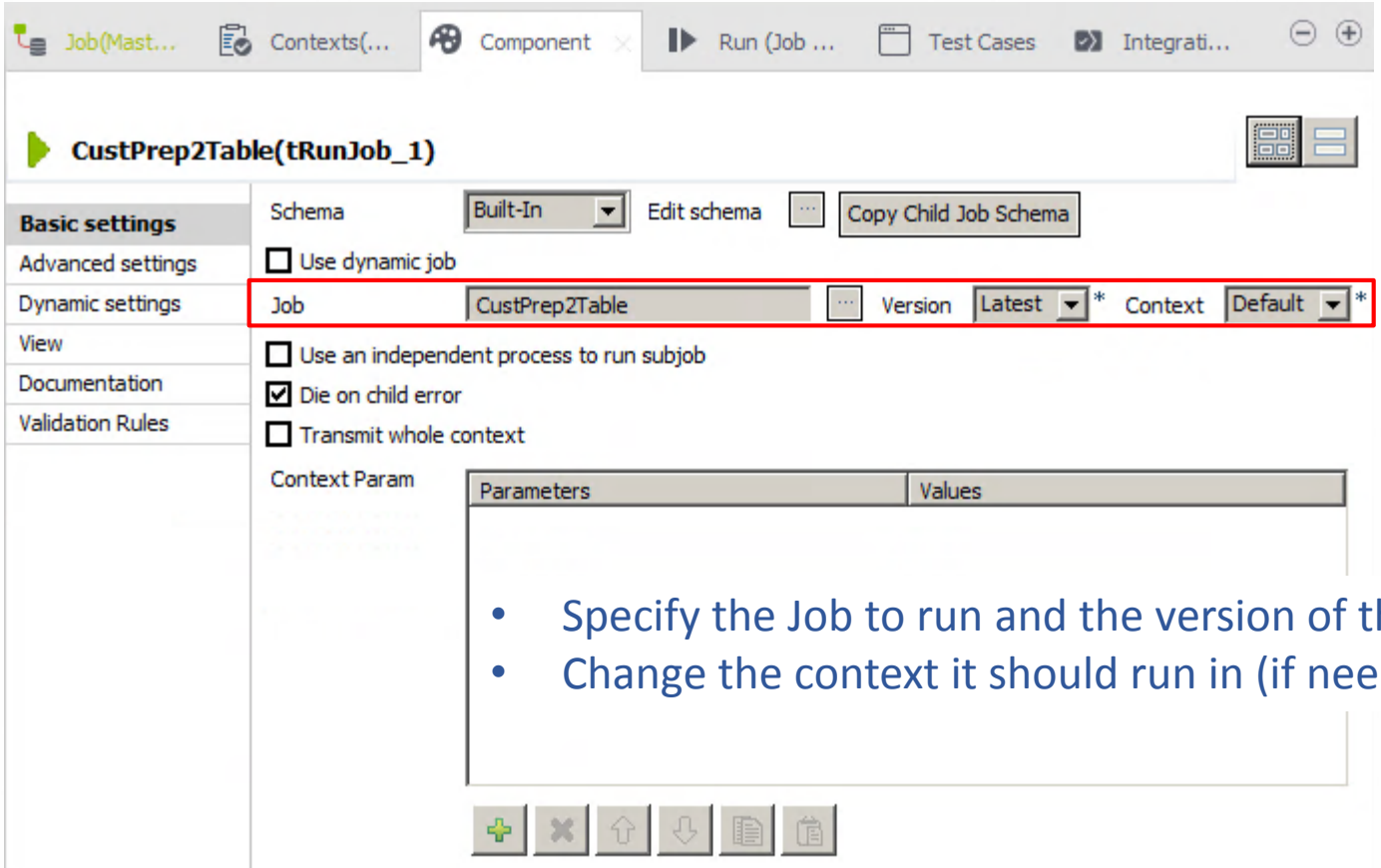
tRunJob – Placement in the design workspace

- tRunJob is placed like any other component
- It can also be dragged/dropped from the Repository



Job Orchestration - Master Jobs

tRunJob – Basics settings

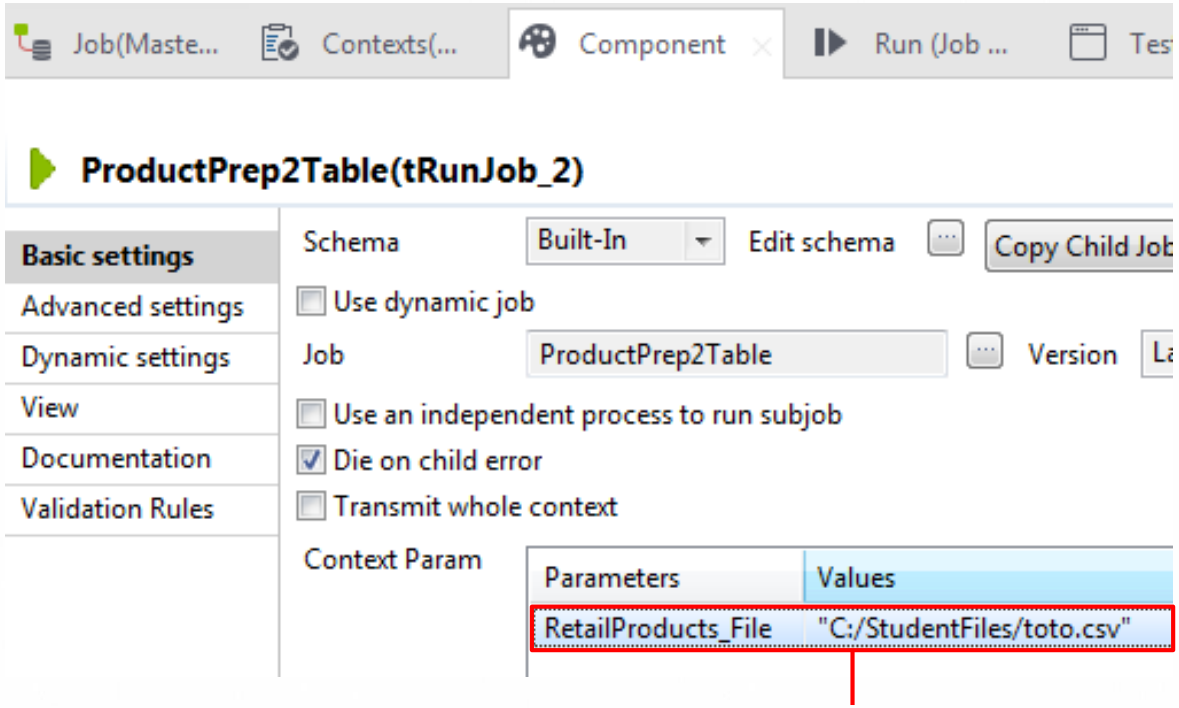


- Specify the Job to run and the version of the Job
- Change the context it should run in (if needed)

Job Orchestration - Master Jobs

tRunJob - Overriding context

The value of **any context variable** can be modified

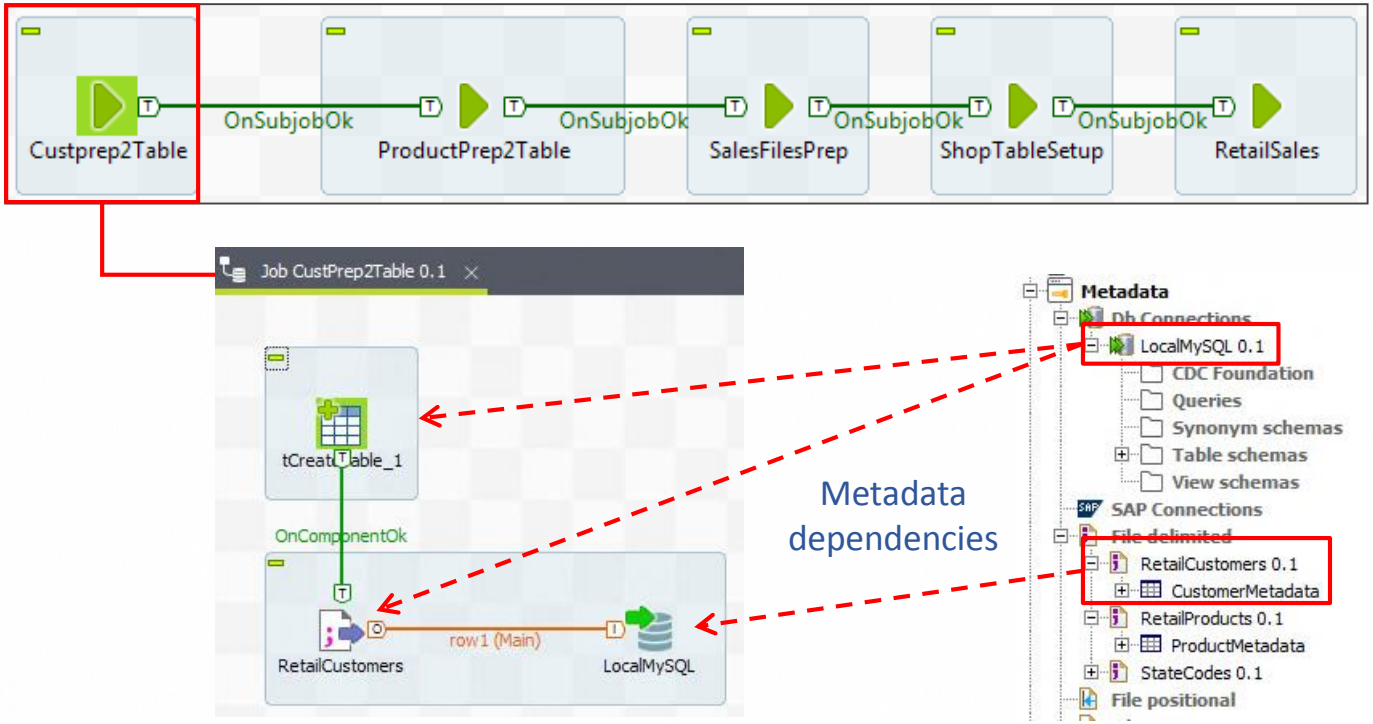


A change here is *only available* for this instance of tRunJob

Exporting Jobs

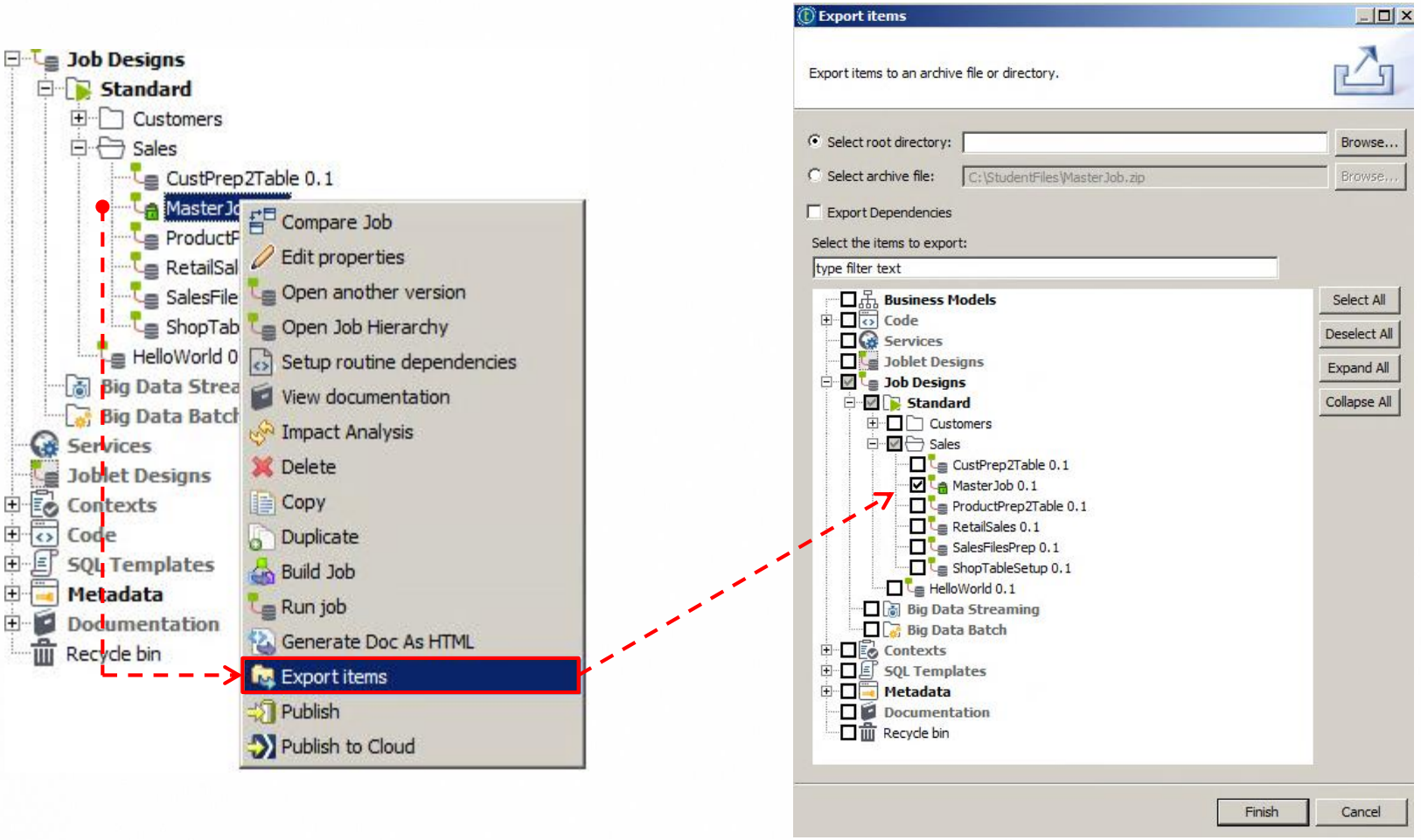
Scenario

You need to share the Master Job with colleagues. It needs to be *packaged with all dependencies* in order to make it work in another Talend Studio instance:

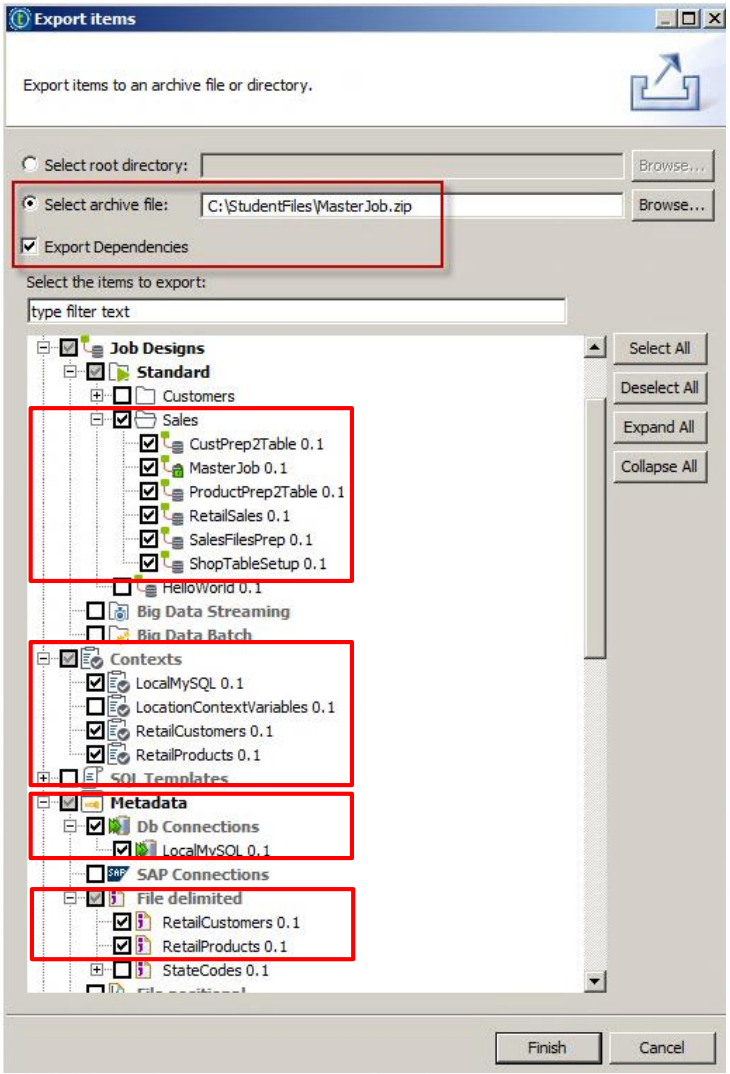


Exporting Jobs

Jobs can be exported from the Repository



Exporting Jobs

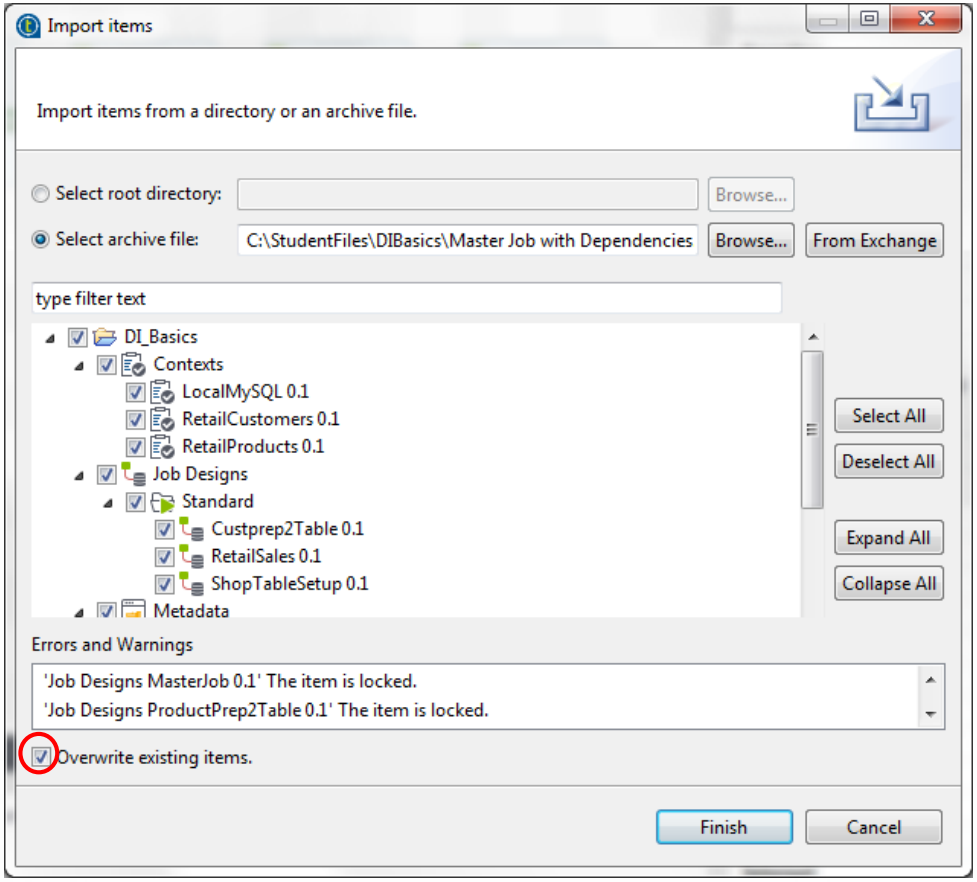


Jobs can be exported:

- As a .zip file
- With all dependencies

Importing Jobs

Jobs can be imported from the Repository

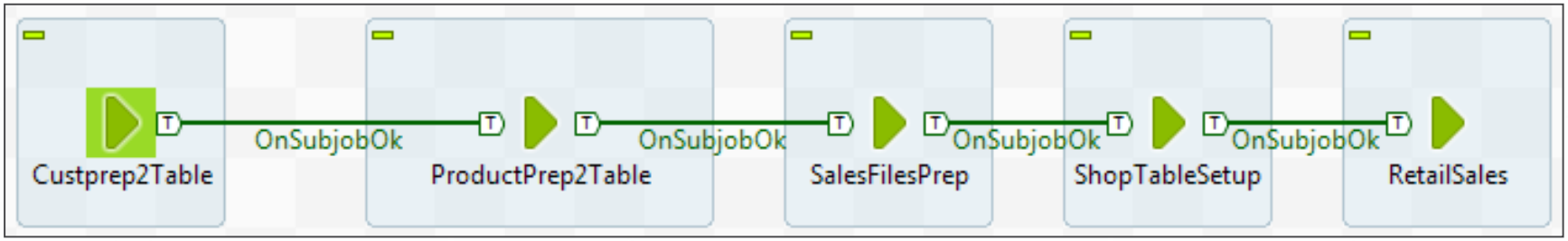


- **Select/unselect items** prior to import
- Select **Overwrite** as needed to clear Errors and Warnings (e.g. if some items already exist in the Repository)
- **Finish** is not actionable until Errors are resolved

Lab overview

Master Jobs

- You will create a Master Job to control the execution of five other DI Jobs.



- Then, you will export the Job and all its dependencies.

Lesson summary

- **tRunJob** is a key component to orchestrate Jobs
- Triggers connect Subjobs based on a condition
  - On Subjob Ok
  - On SubJob Error
- Use Export Items/Import Items to share work with others

Web Services

DI Jobs that work with Web Services

Objectives

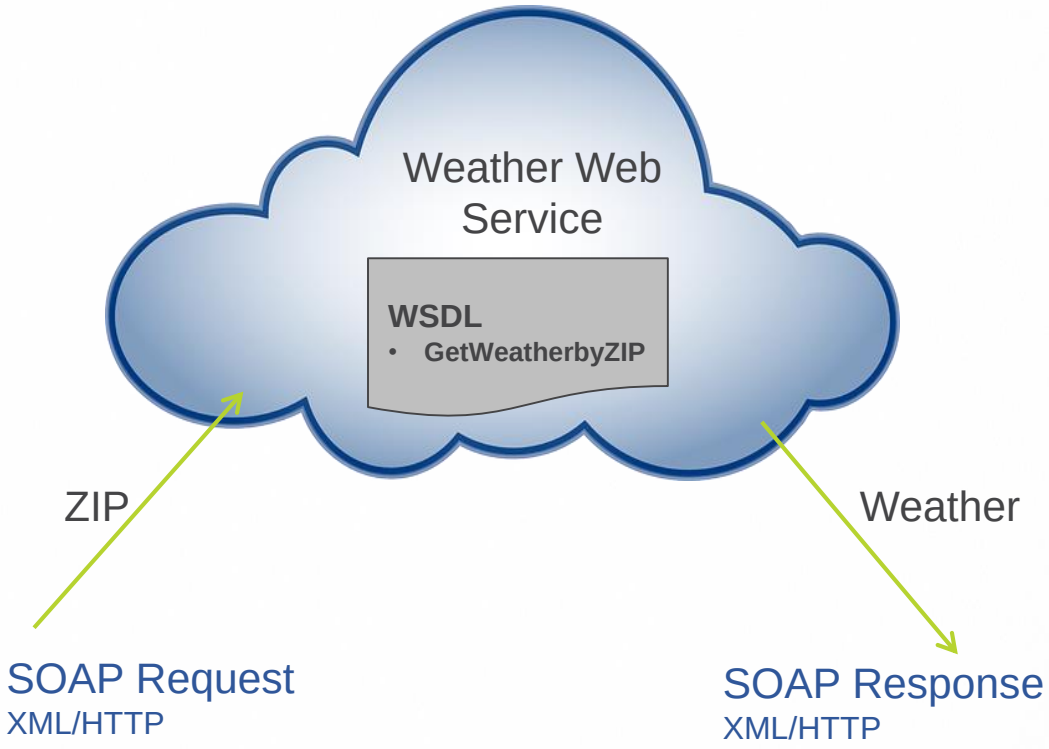
By the end of this lesson, you will be able to:

- Use a Talend component to access a Web Service method
- Extract specific elements from a Web Service reply
- Store Web Service WSDL Schemas for use in a tXMLMap component

Web Services

What are Web Services?

- Web Services are applications that communicate over a network
- Web Services offer a standard way of interoperation for different platforms and frameworks
- Web Services are made available through a network endpoint

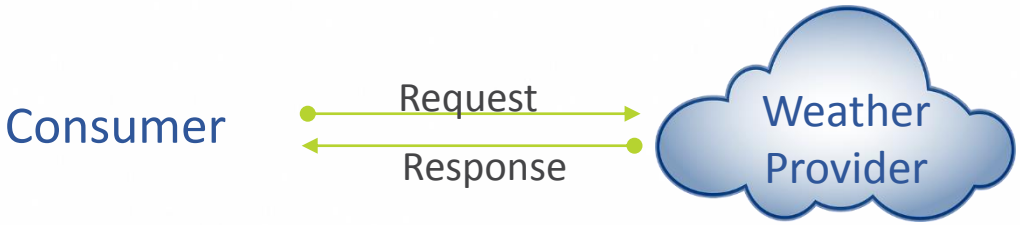


Web Services use messages to exchange request/response info

Web Services Description Language

What is WSDL?

- WSDL is a contract between the Web Service *provider* and *consumer*



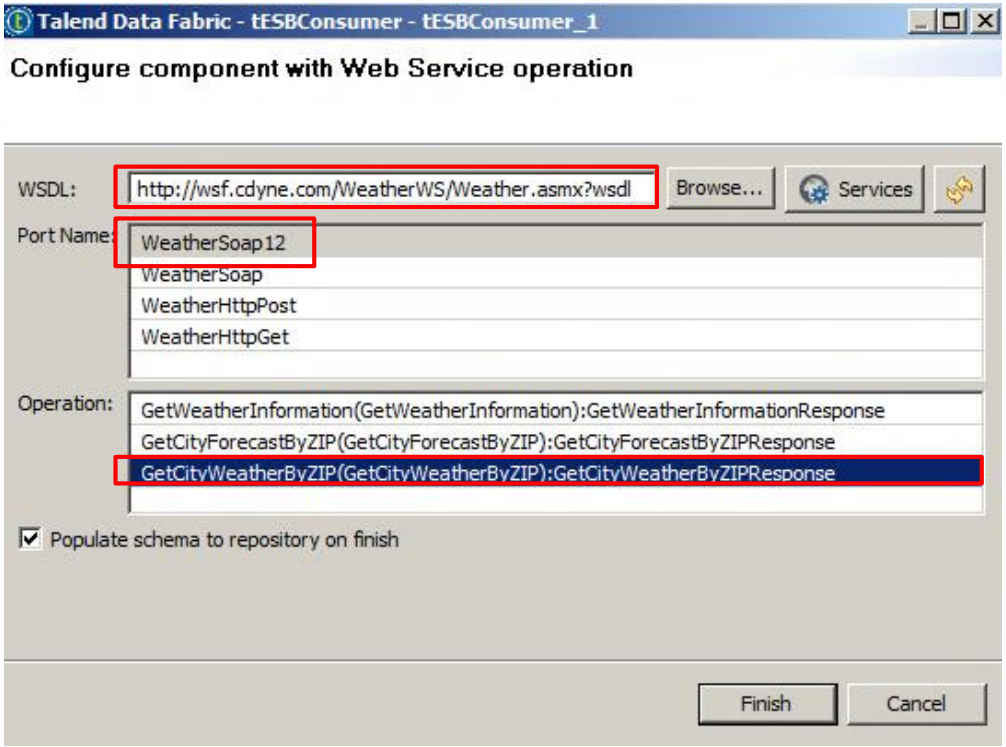
- WSDL defines what operations are supported by the provider
- WSDL is critical to SOAP based Web Services
- In a Talend Job, operations on a Web Service can be invoked with a **tESBConsumer** component

tESBConsumer

Example component configuration for the *request*

Configure the consumer to issue a SOAP request to get city weather by ZIP code

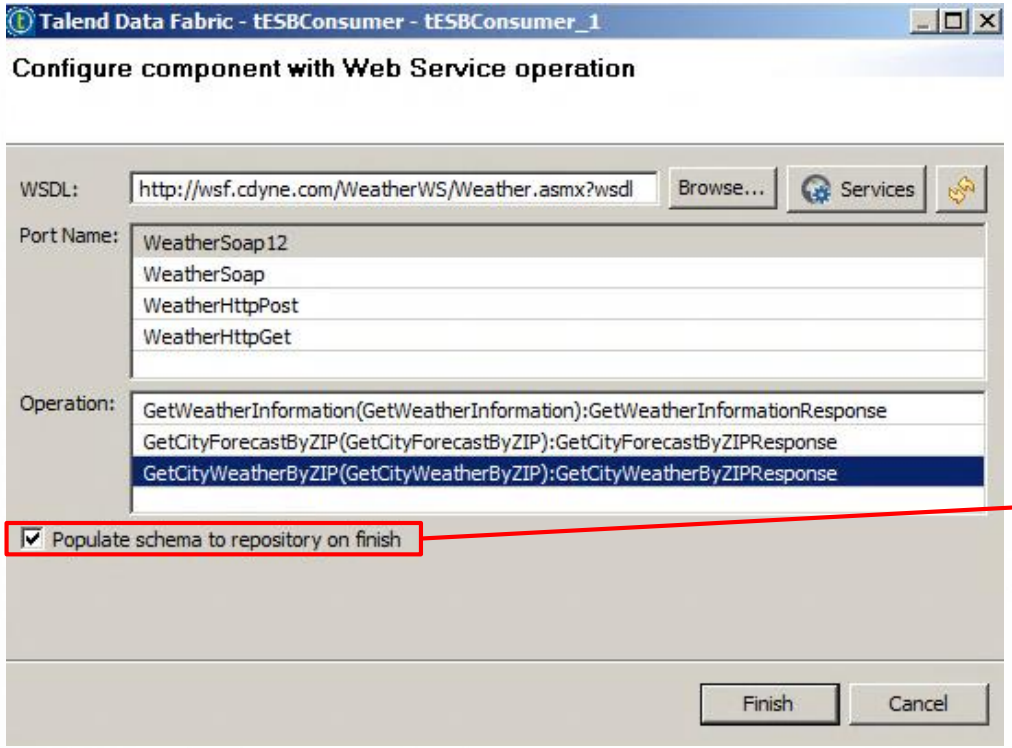
- Enter the **WSDL** URL
- Select **Port Name**
- Select **Operation**

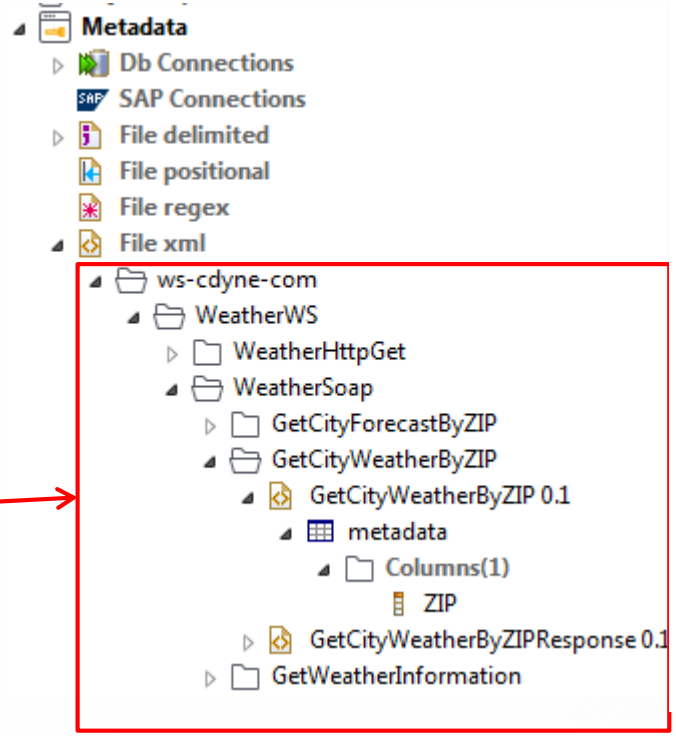


tESBConsumer

Example component configuration for processing the *response* schema

- Although the schema of the Web Service response can be complex, the **tESBConsumer** component allows you to retrieve the schema and save it in the Repository



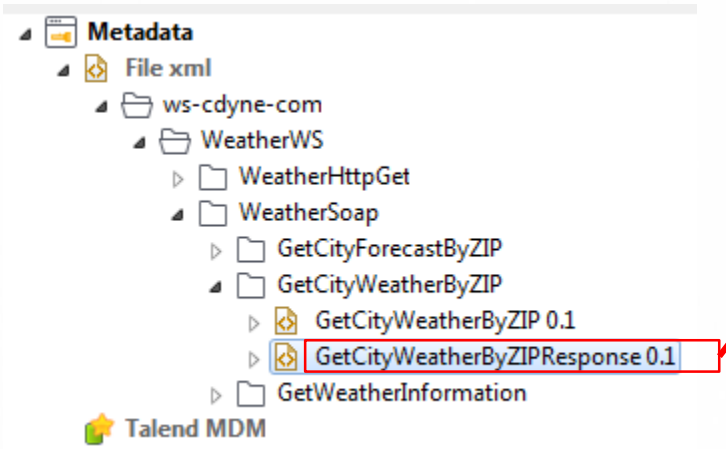


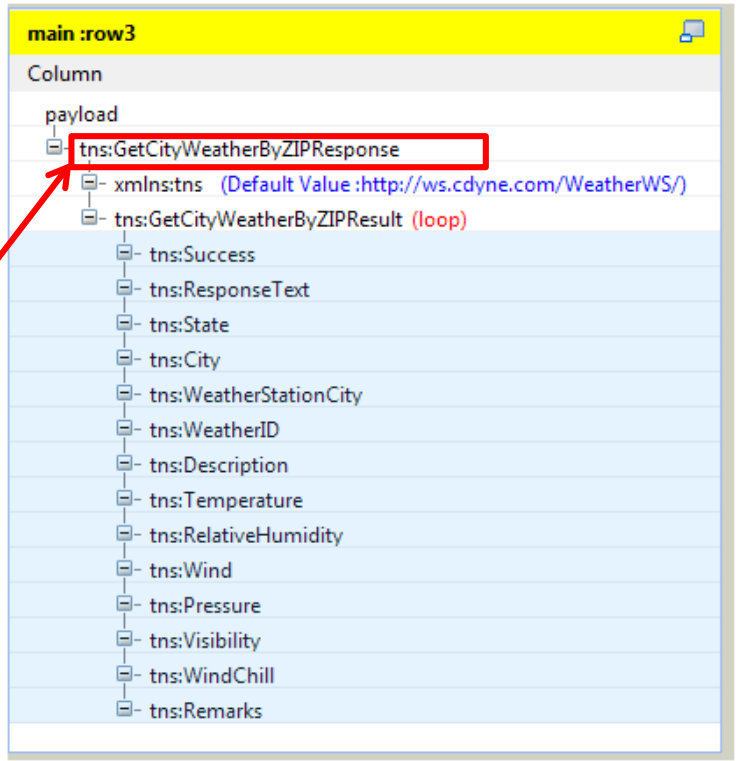
tXMLMap (1 of 2)

Process the XML Web Service response

Use the **tXMLMap** component to extract useful information from the response

- Use the response schema to populate the Input table of the **tXMLMap** component





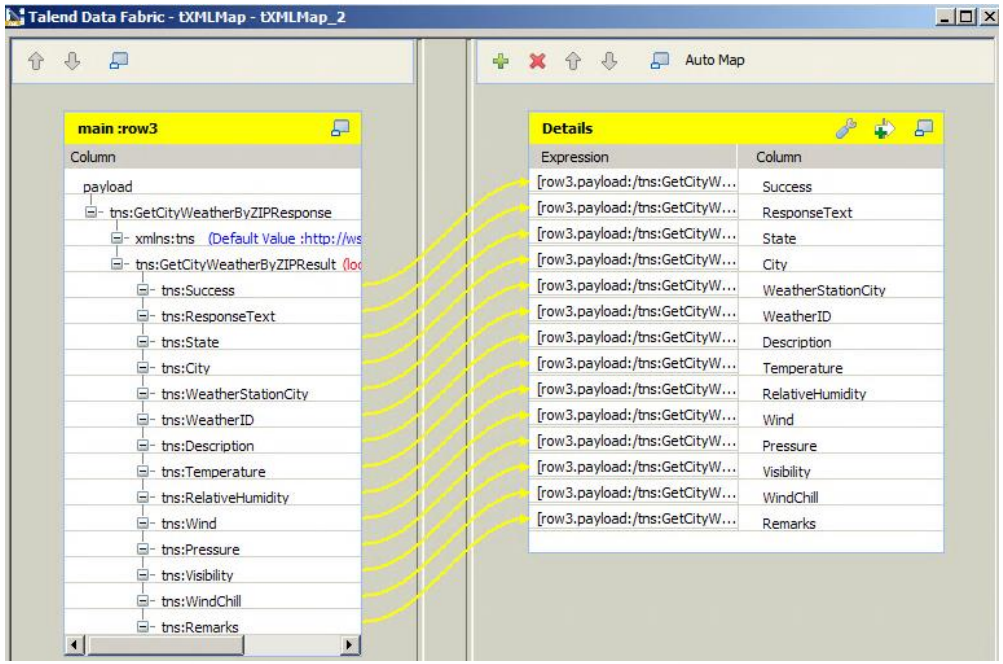
tXMLMap (2 of 2)

Process the XML Web Service response

The **tXMLMap** editor is similar to tMap's

The tXMLMap editor is similar to tMap's when processing the response data flow

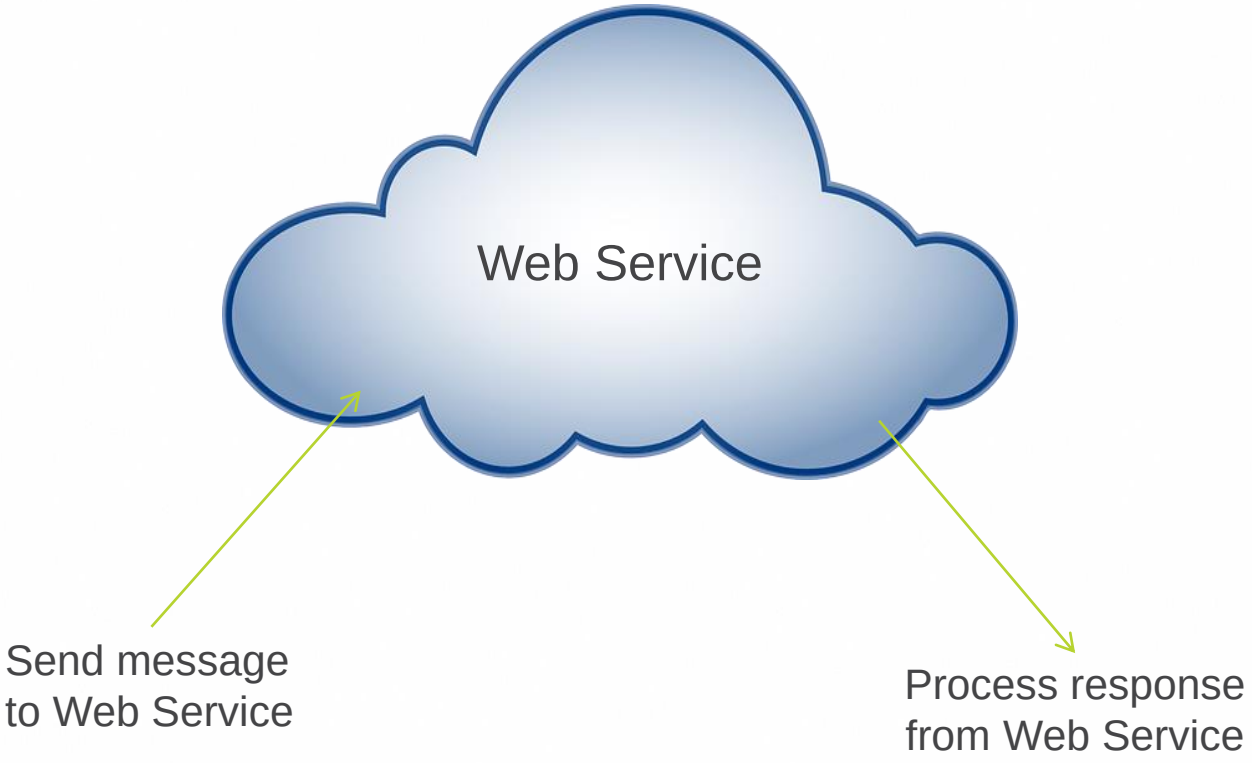
- Route and transform data from/to one or more data sources



Lab overview

Web Services

- You will create a DI Job that accesses a public SOAP based Web Service.



Lesson summary

- In addition to reading and writing to and from files, databases, and other sources, **Talend also supports accessing Web Services**
- **tESBConsumer** is a key component for accessing Web Service providers
- **tXMLMap** can be used to transform the XML response in the data flow

Standalone Jobs

Build and run Jobs away from the studio

Objectives

By the end of this lesson, you will be able to:

- Build and export a Job
- Run an exported Job independently from the Talend Studio
- Create a new version of an existing Job

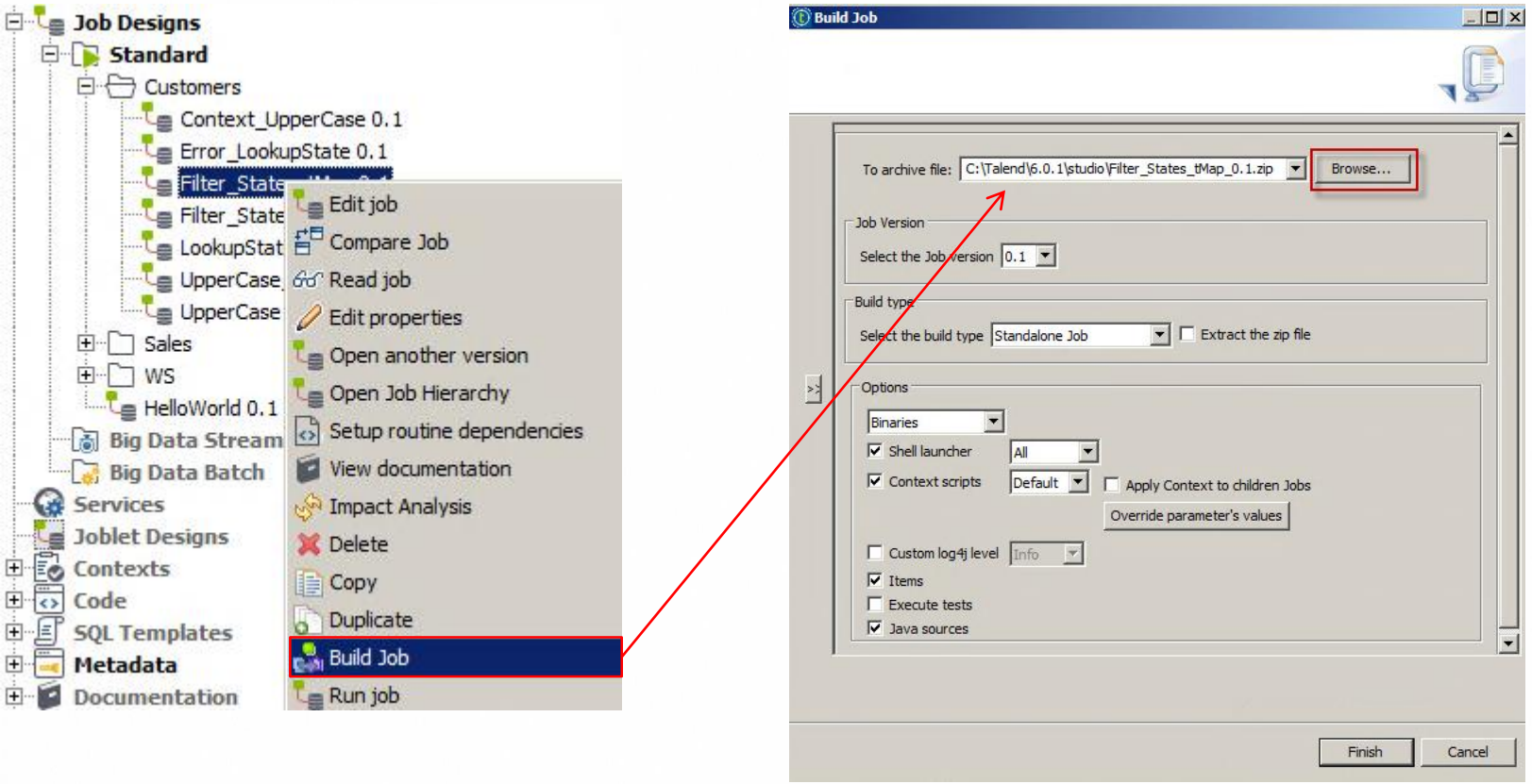
Building a Job

- Export Job
  - Export a Job and its dependencies
  - *Only* usable within the Studio

- **Build Job**
  - Build and deploy a Job
  - Run on *any* Server (independent of Studio)

Building a Job

- Pick the Job in the Repository and select **Build Job**

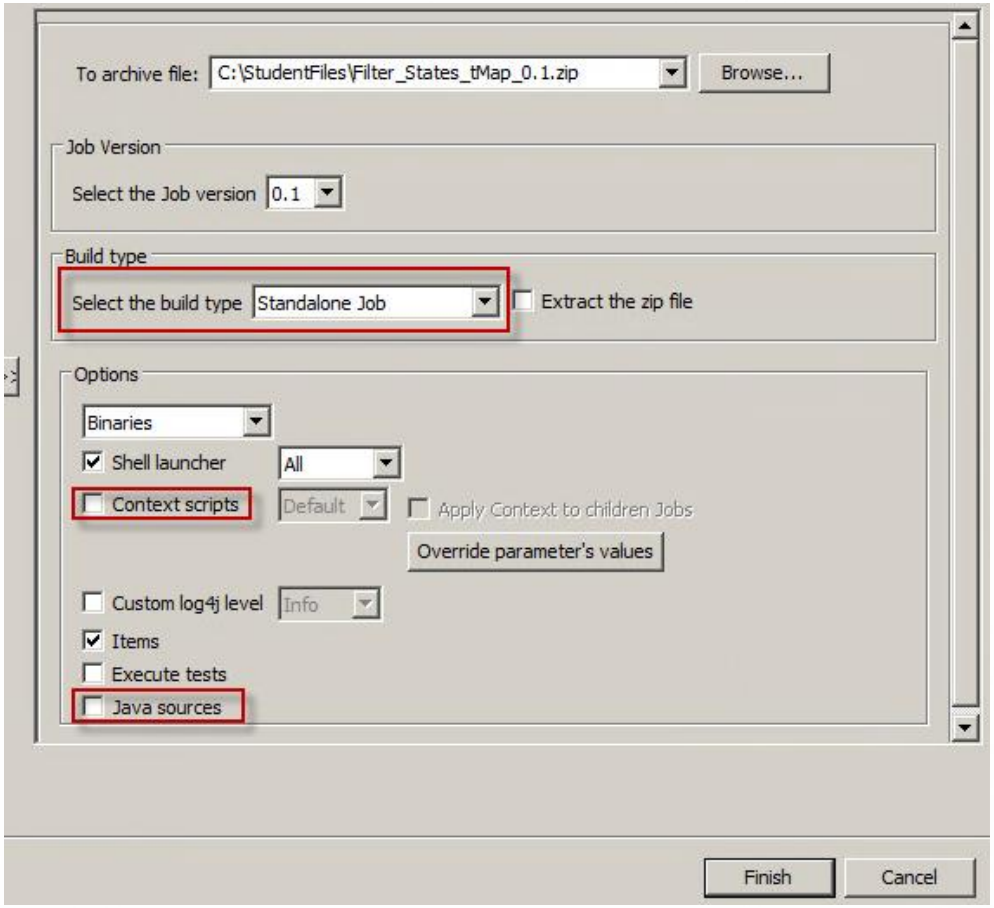


Building a Job

- Select the Build type

Select **Standalone Job**. The following will be included in the built .zip archive:

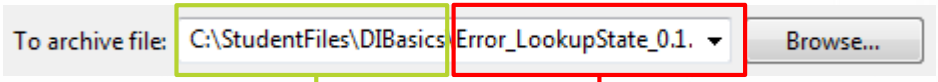
- **JAR files**
  - Java Archive Files
  - Package of files needed for deployment
- ***.bat files -** Windows batch files (for launching)
- ***.sh files** – Unix shell script files



Extract and Run

Extract

- Go to Archive folder
- Extract .zip

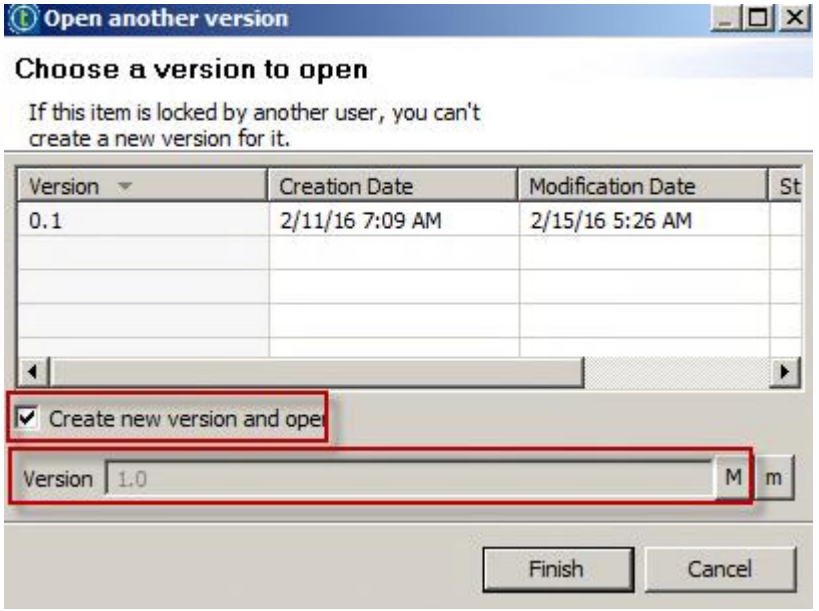


Run

- Invoke executable
  - Error_LookupState_0.1.bat (Windows)
  - Error_LookupState_0.1.sh (Linux)

Job Versioning

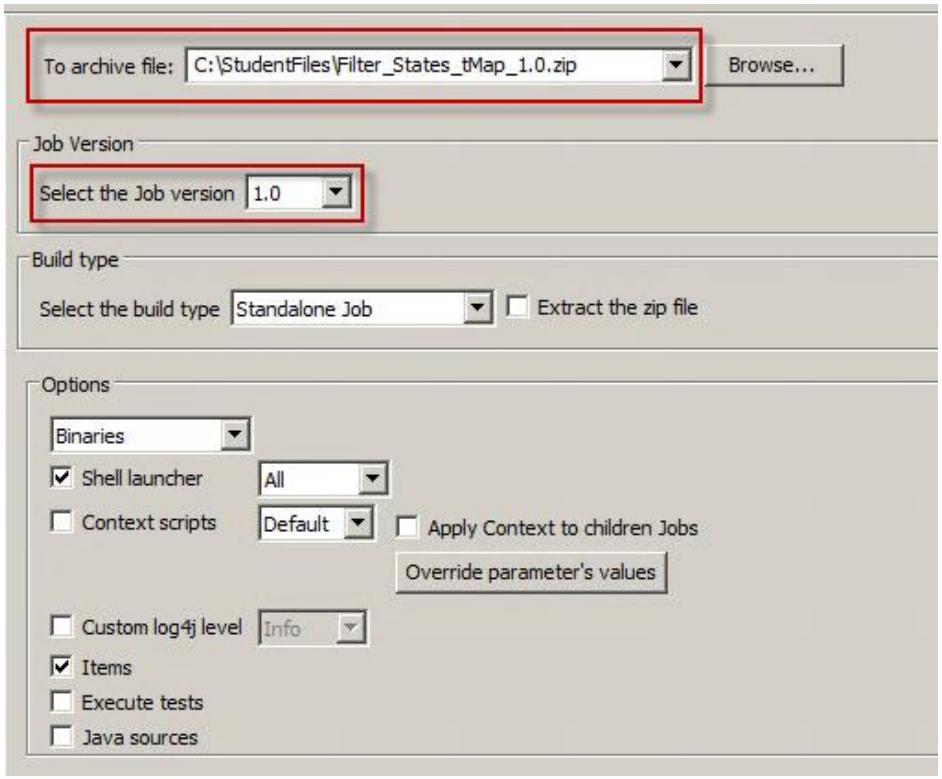
- Studio offers basic version control (**M**ajor.**m**inor)
  - Create Job
  - Open another version



Increment **M**ajor/**m**inor

Job Versioning

- Studio offers basic version control (**M**ajor.**m**inor)
  - Build another version



Lab overview

Stand alone Jobs

- Build Jobs that can be run outside of the Talend Studio
- Use basic version control

Lesson summary

- **Build Job** is different than **Export Job**
- Built Jobs can be run outside of the Studio
- Basic version control is easy to implement
  - Create new Job
  - Open another Version
  - Build Job

Best Practices & Documentation

Plan – Develop - Document

Objectives

At the end of this lesson, you will be able to:

- Name a Job and fill out its properties using best practices
- Provide additional information for Job components
- Generate and view HTML documentation for your Job

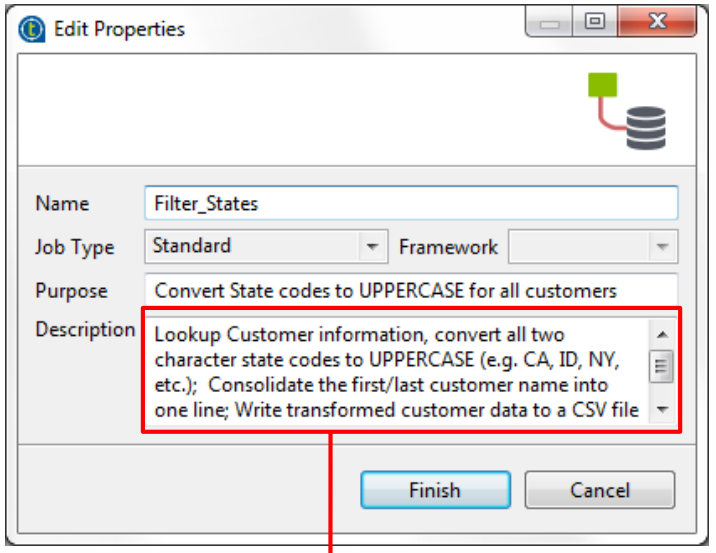
Scenario

- There are a lot of inconsistencies and no documentation on Jobs developed by your team.
- Your new manager wants to implement several best practices for the development team. Most are documentation related, from Job and component naming conventions to generating HTML documentation for others departments to view.

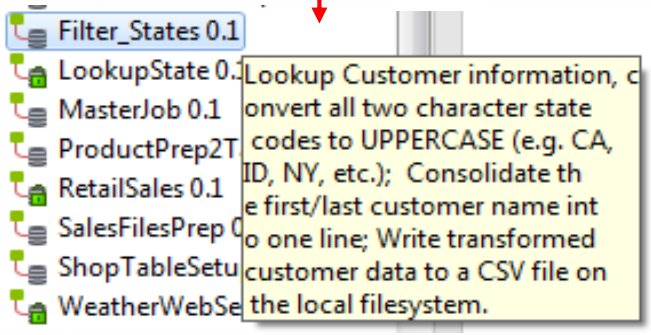
Best Practices - Jobs

Name, purpose and description are all key properties that help document your Job

- Create a New Job
  - **Name** – Recommend a naming convention
  - **Purpose** – OK to keep brief
  - **Description** – More detailed, shows in repository hover-text
- Modify anytime with **Edit Properties** (Right-click Job in repository)



Repository

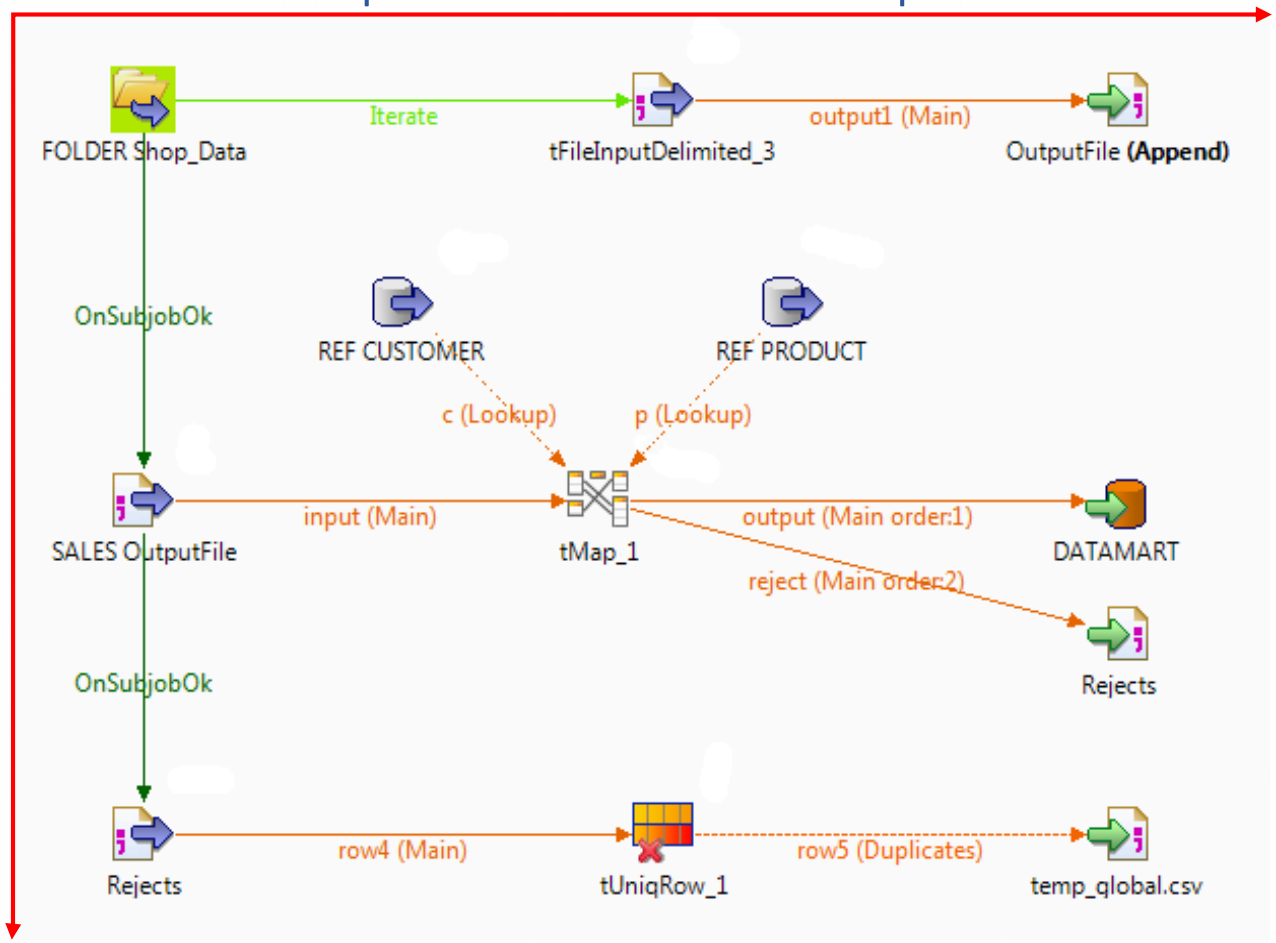


Best Practices - Jobs

Guidelines for Job layout

Work from the top. Place and connect components from left to right...

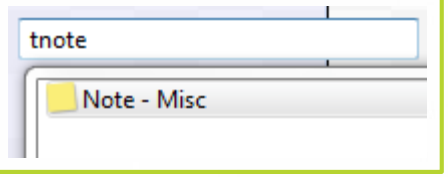
... and top to bottom



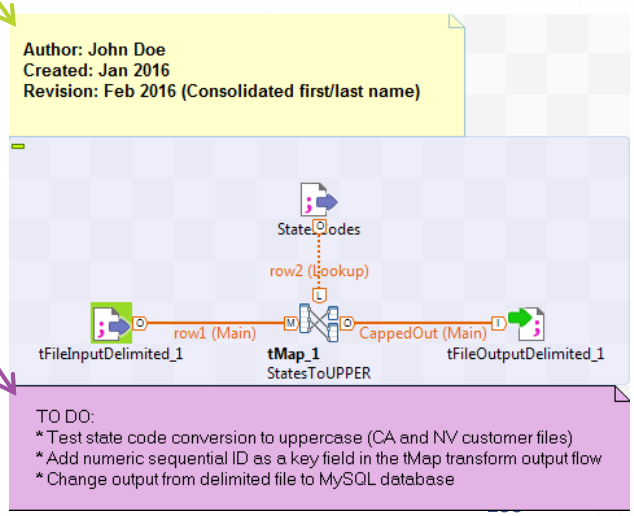
Best Practices - Jobs

Use a **Note** component to help document your Job in the design workspace

- Place a **Note** in the design workspace
- Possible format and content conventions:
  - Author: John Doe
  - Created: Jan 2016
  - Revision: Feb 2016 (Consolidated first and last name)
  - To Do list: Test state code …



- Explore basic format options
  - Font, size, **bold**, *italics*
  - Opacity
  - Colors (text, border, fill)
  - Vertical/Horizontal alignment



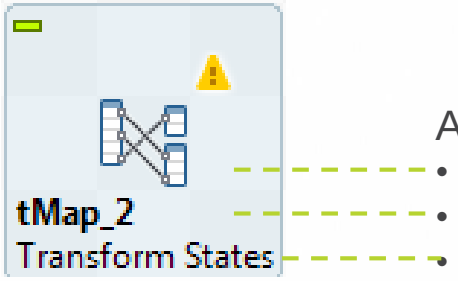
Best Practices - Components

Customizing the component label improves readability in the design workspace

Field	Description	Default Value
Label format	Text label shown in design workspace	__UNIQUE_NAME__

Customized example:

```
<b>__UNIQUE_NAME__</b><br>Transform States
```



At a glance:

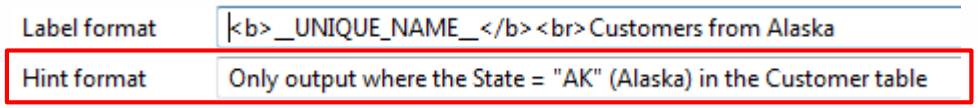
- Icon (develop familiarity)
- Component Name (in bold)
- Primary function

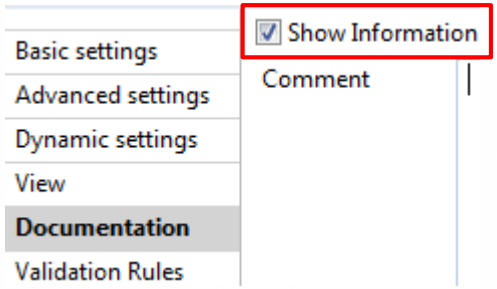
Best Practices - Components

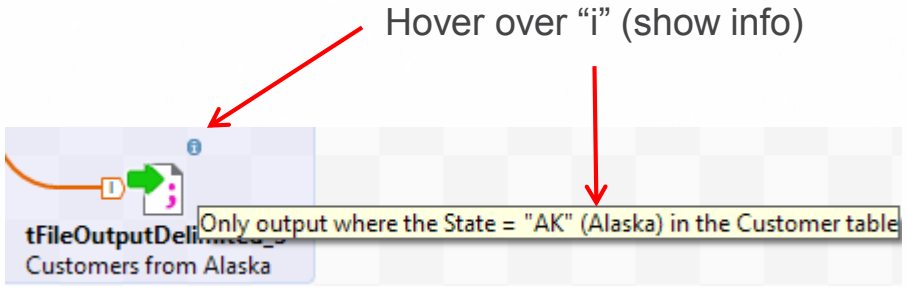
The Hint format provides helpful component information in the workspace

Field	Description	Default Value
Hint format	Hidden tooltip. Visible when hover-over component *and* Documentation "show information" selected	__UNIQUE_NAME__ __COMMENT__

- Documentation > Show information



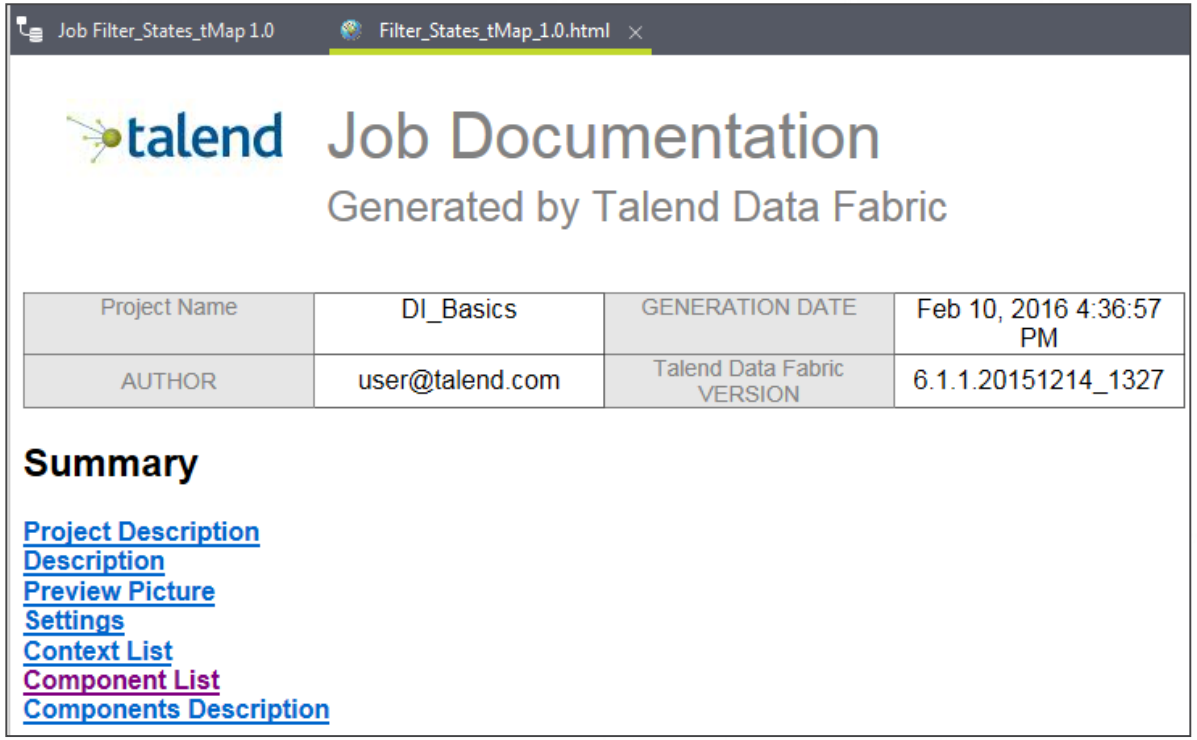




View Documentation

Use View documentation in the Studio during the development phase of your Job

- Repository > *JobName* > View documentation
  - Opens a tab with HTML documentation for review

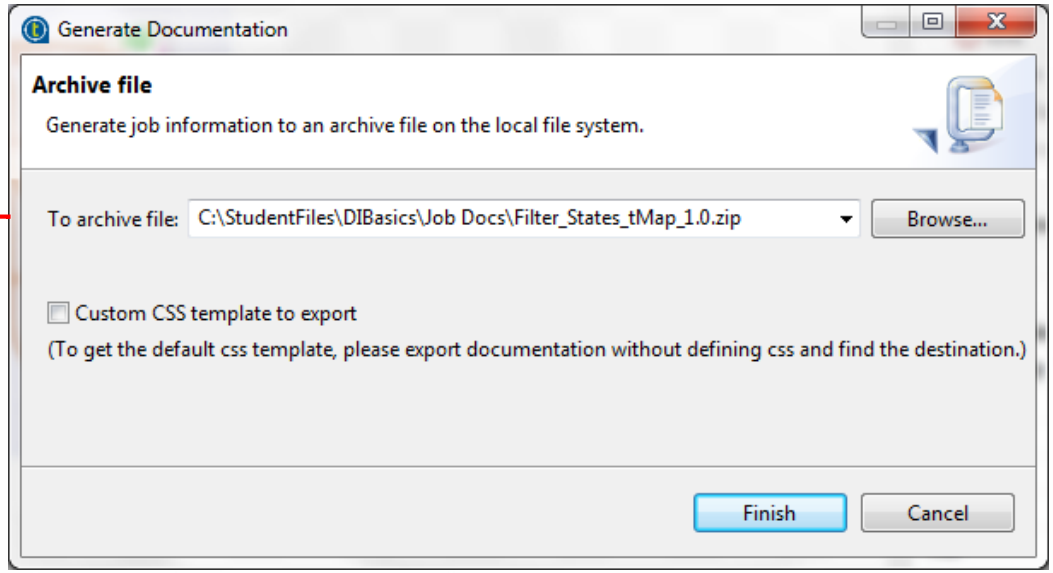


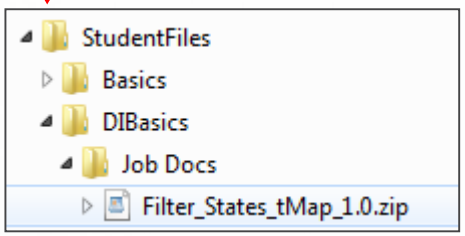
Actionable hyperlinks with additional content

Generate Documentation as HTML

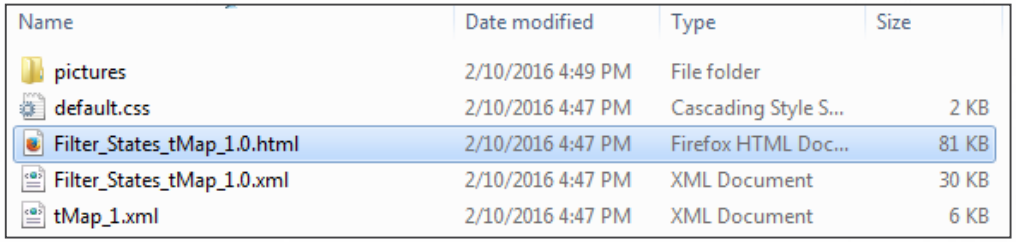
Once the development phase for your Job is completed, generate the documenation

Repository > *JobName* > Generate Doc as HTML





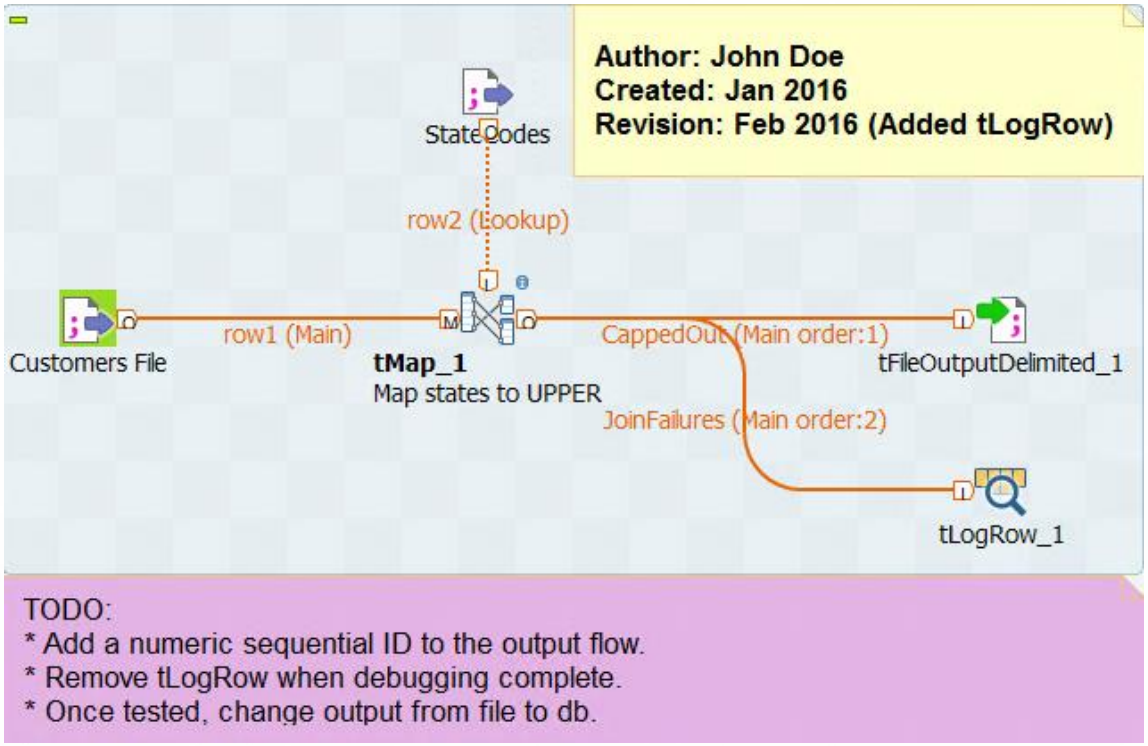
Unzip



Lab overview

Best Practices and Documentation

- Use several best practices while developing a Job
- View HTML Job documentation in the Studio
- Generate and view documentation external to the Studio



Lesson summary

- There are several best practices to utilize during the development of a Job that help with the overall Job documentation
  1. Planning
     - Business Model
  2. Development
     - Naming conventions, labels and hints
  3. Documentation
     - Generate and View HTML