

## **Lab Center – Hands-on Lab**

### **Session 7461**

## **Add Intelligence to Business Automation with IBM Business Automation Insights**

**Christophe Jolif**, IBM Digital Business Automation, Architecture & Development,  
[christophe.jolif@fr.ibm.com](mailto:christophe.jolif@fr.ibm.com)

**Sebastian Carbajales**, IBM Digital Business Workflow, Architecture & Development,  
[sebastia@ca.ibm.com](mailto:sebastia@ca.ibm.com)

# Table of Contents

<b>Disclaimer</b>	3
<b>Overview</b>	5
Lab Introduction	5
The Recommendation Service Scenario	5
Overview of the Solution	6
<b>Lab Environment</b>	8
Setting up a Watson Machine Learning Service Instance	8
Prerequisites	9
<b>Lab Setup</b>	10
Start the IBM Business Automation Workflow Server	10
Start the Jupyter Environment	10
<b>Step by Step Instructions</b>	12
1    Overview of the Car Loan Approval Process	12
1.1    Open the Car Loan Process Application	12
1.2    Examine the Car Loan Approval Process	12
1.3    Examine the Loan Approval Task UI	15
2    Train and Deploy a Machine Learning Model to provide Loan Approval Recommendation	18
3    Add Recommendation to the Car Loan Approval Process	19
3.1    Implement the Watson Scoring API Call	19
3.1.1    Examine the External Service	19
3.1.2    Update the Invoke Recommendation ML Model Service Flow	21
3.2    Modify the Approval User Interface to Display the Recommendation	25
3.3    Call the Machine-Learning Model to Obtain a Prediction	28
3.4    Test the Application	34
<b>Summary</b>	38
<b>We Value Your Feedback!</b>	39
<b>Appendix A: Jupyter Notebook</b>	40

## Disclaimer

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision.

The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract.

The development, release, and timing of any future features or functionality described for our products remains at our sole discretion I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve results like those stated here.

Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information. **This document is distributed "as is" without any warranty, either express or implied. In no event, shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.**

IBM products and services are warranted per the terms and conditions of the agreements under which they are provided.

IBM products are manufactured from new parts or new and used parts.

In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply."

**Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.**

Performance data contained herein was generally obtained in controlled, isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

It is the customer's responsibility to insure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and

regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer follows any law.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products about this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. **IBM expressly disclaims all warranties, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a purpose.**

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

IBM, the IBM logo, ibm.com and [names of other referenced IBM products and services used in the presentation] are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at: [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

© 2019 International Business Machines Corporation. No part of this document may be reproduced or transmitted in any form without written permission from IBM.

**U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.**

# Overview

In this lab, you will learn how to leverage a machine-learning model to optimize the decision taken in a business process.

Artificial intelligence (AI) can be combined with business processes management in many ways. For example, AI can help transform unstructured data into data that a process can work with, through techniques such as image processing or natural language processing; assistants and bots can be integrated with processes to provide a richer user experience. Processes themselves produce a large amount of data as they execute. This historical data can be used to optimize the process by using machine learning techniques. For example, a recommendation could be provided for a decision that needs to be taken as part of the business process based on past similar situations.

The focus of this lab is to provide such a recommendation for a simple process.

## Lab Introduction

In this lab you will leverage process data captured by **IBM Business Automation Insights**, a platform-level component that provides visualization insights to business owners and that feeds a data lake to infuse artificial intelligence into **IBM Digital Business Automation**. You will use the data to build your own custom machine learning model. You will then integrate this model into an existing process, using **IBM Business Automation Workflow**, to provide a recommendation to your process participants.

In this lab, you will learn how to:

- Load time series data, from the **IBM Business Automation Insights** data lake, for a specific tracking point in the Workflow process
- Explore the format of the data and interpret it
- Create an Apache® Spark machine learning pipeline to process the data
- Train and evaluate the recommendation model
- Persist a pipeline and model into an IBM Watson Machine Learning repository
- Deploy a model for online scoring using Watson Machine Learning Client API
- Score sample scoring data by using the Watson Machine Learning API
- Use an External Service to invoke the recommendation model and provide a decision recommendation in a Workflow Coach

## The Recommendation Service Scenario

In this lab you will be working with a simple **Car Loan Approval** process.

Imagine a financial institution which has set up a Workflow process to handle car financing requests. Some of loan requests are simple to process because, typically, the amount request is small and the

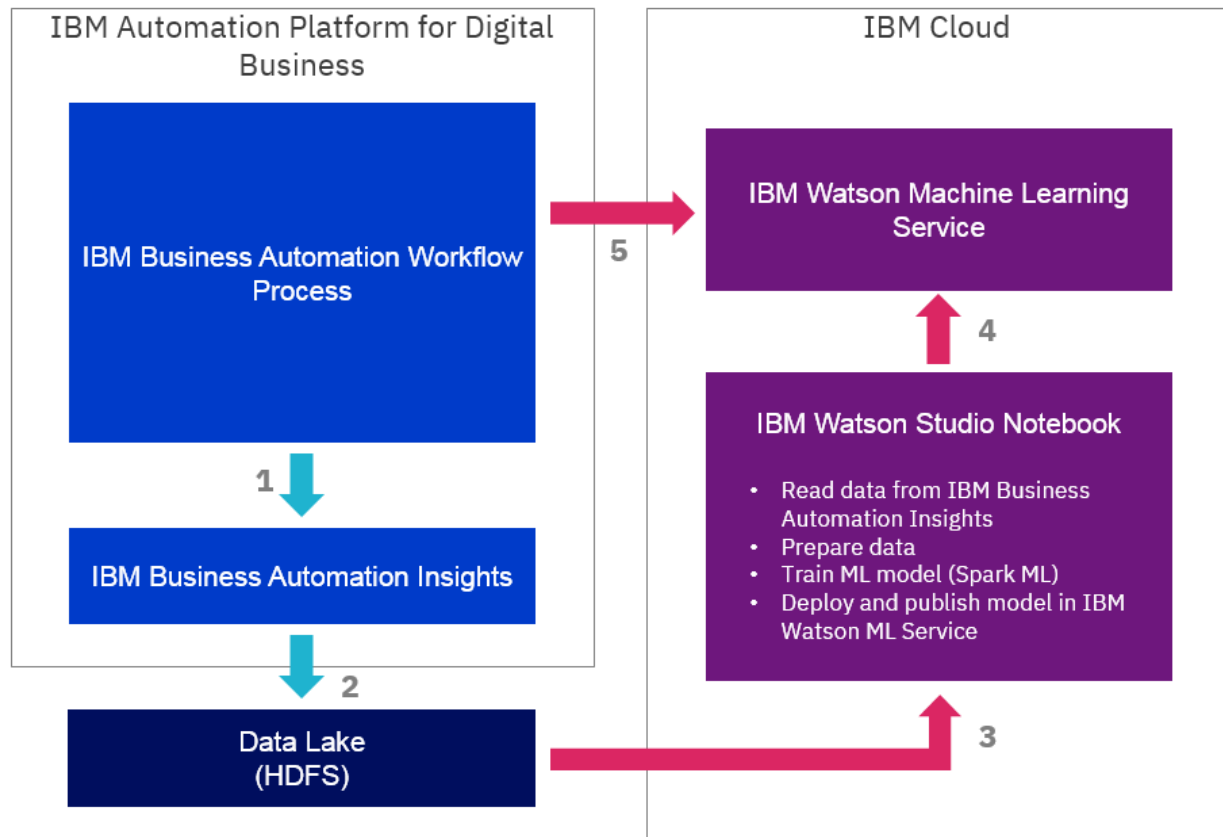
customer's credit score is high and/or the subject vehicle is newer and in demand, thus with a low risk of defaulting. Such loan requests can be approved automatically or, at least, follow a fast approval path. Some financing requests are more complex and, therefore, their approval path may include more steps. Let us further assume that the approval decision or the decision on which path to follow is a human task and this task is captured as a task of a Workflow process. Then it becomes interesting to consider whether a machine-learning algorithm can help figure out which decision to take, based on past decisions.

This scenario can be adapted to any kind of human decision process. In the loan request example, the decision consists of approving or rejecting a vehicle finance application, which boils down to a yes-or-no decision. Such decisions can translate into a binary classification machine-learning problem. However, if the decision consists in dispatching a process into many other subprocesses, the scenario becomes a multiclass classification problem, which Business Automation Insights can also handle.

In this scenario you will work with a simple process implemented to review and approve a loan request. The process takes a loan application, which includes information about the applicant as well as the vehicle, that is reviewed by an underwriter who decides to approve or reject the request. The current process version has been used for a long time and each decision made by the underwriter team has been captured by **IBM Business Automation Insights**. You now want to leverage this data to optimize the work of the underwriter by implementing a service that will provide a recommendation on whether to approve or reject a new loan request.

## Overview of the Solution

The following figure illustrates how all the different elements and cloud services are used together to build the expected recommendation service.



Everything starts with the business process itself running in **IBM Business Automation Workflow**. As the process instances execute, the business data, which in our scenario contains information about the loan request, is captured by the **IBM Business Automation Insights (BAI)** service (1). As the incoming events are processed the data is stored in the data lake configured with BAI (2).

Once the data is captured in **HDFS**, it can be used to train a machine-learning model using an **IBM Watson Studio Notebook** (3). After the model is trained with existing loan request data and approval decisions, it would be able to provide recommendations on whether to approve or reject new loan request.

The trained model needs to be deployed into an **IBM Watson Machine Learning** service (4). This service stores the machine-learning model and provides a scoring.

Finally, the scoring endpoint can be invoked by the **IBM Business Automation Workflow** business-management process and the result used to provide a recommendation on the approval task user interface (5).

# Lab Environment

In order to expedite the execution of this lab several artifacts have been prepared in advance. These will be described throughout the lab.

The complete lab procedure is captured by two documents: this document and a Jupyter notebook. The instructions will indicate when to work with each one.

The following files are provided for this lab:

- A TWX with the **IBM Business Automation Workflow** artifacts required to complete the lab.
- A Jupyter notebook to train and deploy the machine learning model built during this lab

The TWX file has already been imported to the Workflow Center, and therefore these artifacts are ready for you to use.

In order to perform the activities described in this lab, you will **utilize IBM Business Automation Workflow** development time tooling (**Process Designer**) and a **Jupyter** notebook environment. A preconfigured environment has been setup with several VMs. You will work on the 'Workflow' VM only.

Also, an instance of an **IBM Watson Machine Learning** service has been preconfigured to be shared by all lab participants. If you wish you use your own instance you must sign up for an **IBM Cloud** account and deploy one. See details below.

The first action you will take in this lab is to open the Workflow VM and proceed from there. All required environments should be up and running but the Step-by-Step instructions will include steps to start them if needed.

## Setting up a Watson Machine Learning Service Instance

1. Create an IBM id and register for IBM Cloud: <https://console.bluemix.net/registration/>
2. Then log in to IBM Cloud and create an instance of Watson Machine Learning from the IBM Cloud catalog: <https://cloud.ibm.com/catalog/services/machine-learning>
3. Create a new service credential
  - i. Open the service instance
  - ii. Select **Service Credentials** on the left panel
  - iii. Click the **New credential** button
  - iv. Accept the defaults and click **Add**
  - v. Once the credentials are created, click **View credentials** under the **Actions** column
  - vi. Copy the JSON object and save for later use



## Prerequisites

Although a preconfigured environment will be provided, the details are included here for completion.

To be able to complete the lab exercise the following must be installed:

- IBM Business Automation Workflow
- IBM Business Automation Insight
  - Business Automation Insight must be installed and connected to an HDFS data lake.
- IBM Watson Machine Learning service
- Jupyter notebook environment
  - IBM Watson Studio can be used if the HDFS data lake is accessible by it
  - The notebook requires a Python 3.5 and Spark 2.1 kernel to run. Watson Machine Learning only support Spark 2.1.

Once you have the installed the various elements, ensure you have:

- Credentials for your **IBM Business Automation Workflow** instance
  - Preconfigured environment: **deadmin/Think4me**
- Credentials for the HDFS server used by **IBM Business Automation Insights**
  - Preconfigured environment: **admin/admin**
- IBM Watson Machine Learning service credentials. See [here](#).

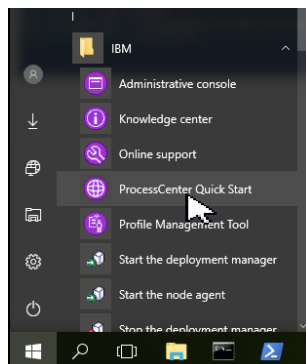
## Lab Setup

This section describes the steps required to get the environment up and running, if needed. Start by logging into the **'workflow'** VM in your environment. When the browser window opens for the VM, if you don't see the full screen (including the tool bar at the bottom) you can click on the **Fit to Window** button to adjust the resolution

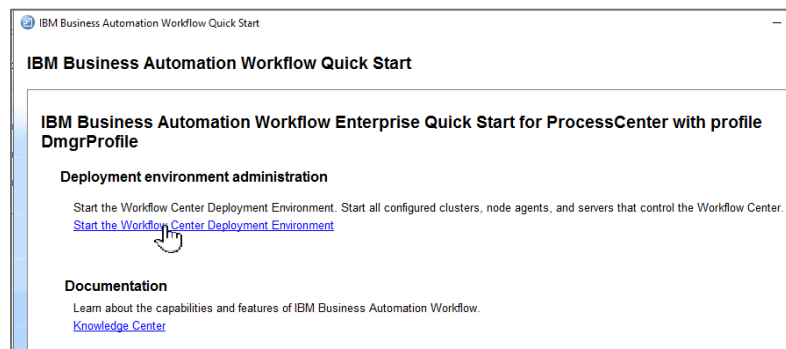


### Start the IBM Business Automation Workflow Server

- \_\_\_ 1. Open the **ProcessCenter Quick Start** via Start > IBM > ProcessCenter Quick Start.



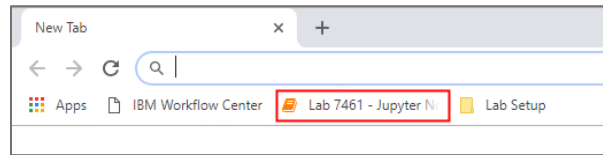
- \_\_\_ 2. Ensure the Deployment Environment has been started. If not, click on the **Start the Workflow Center Deployment Environment** link.



It may take a few minutes for the system to be fully operational. You can monitor the dialog above to find out when the environment is up.

### Start the Jupyter Environment

- \_\_\_ 3. Check that the Jupyter environment is up by opening the Chrome browser and clicking on the **Lab 7461 – Jupyter Notebook** link in the bookmark bar.



If the notebook fails to load proceed to the next step.

- \_\_\_ 4. Open a new Windows PowerShell from the task bar.



- \_\_\_ 5. Run '**.\LaunchJupyter.bat**' script to start the environment.

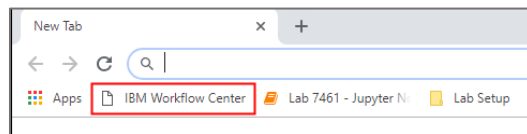
# Step by Step Instructions

## 1 Overview of the Car Loan Approval Process

In this section you will examine the as-is process to understand how it can be improved by injecting a recommendation service. The process, and supporting artifacts, are contained in the **Lab 7461 – Car Loan Approval (LA)** application.

### 1.1 Open the Car Loan Process Application

- \_\_\_ 1. Open the **IBM Workflow Center** by launching the Chrome browser and clicking on the corresponding link in the bookmark bar



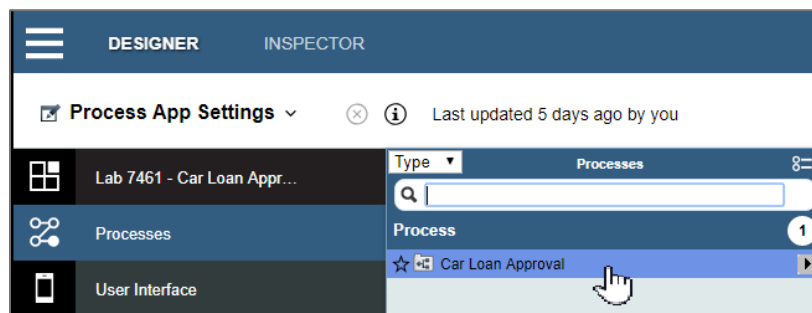
- \_\_\_ 2. Log in as user/password **deadadmin/Think4me**

- \_\_\_ 3. Next to the **Lab 7461 - Car Loan Approval** application click **Open in Designer**

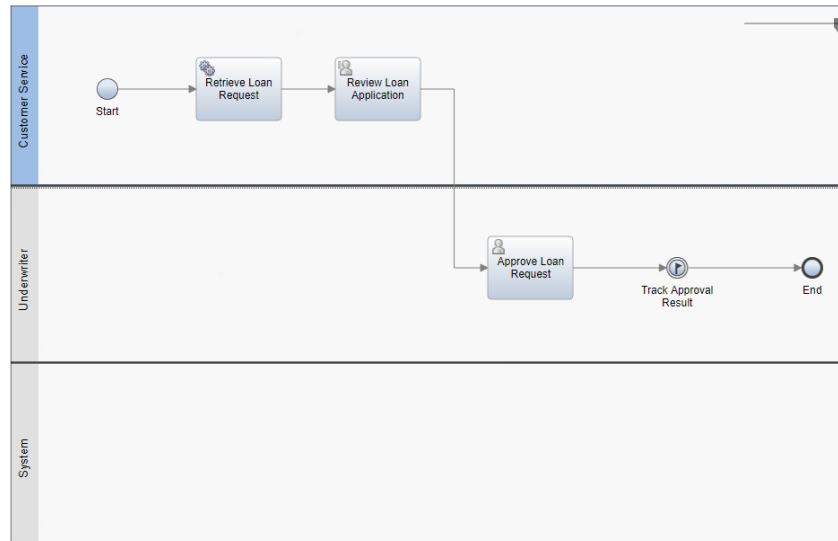


### 1.2 Examine the Car Loan Approval Process

- \_\_\_ 6. Open the **Car Loan Approval** process: **Processes** category > **Car Loan Approval** process



- \_\_\_ 7. Examine the process structure



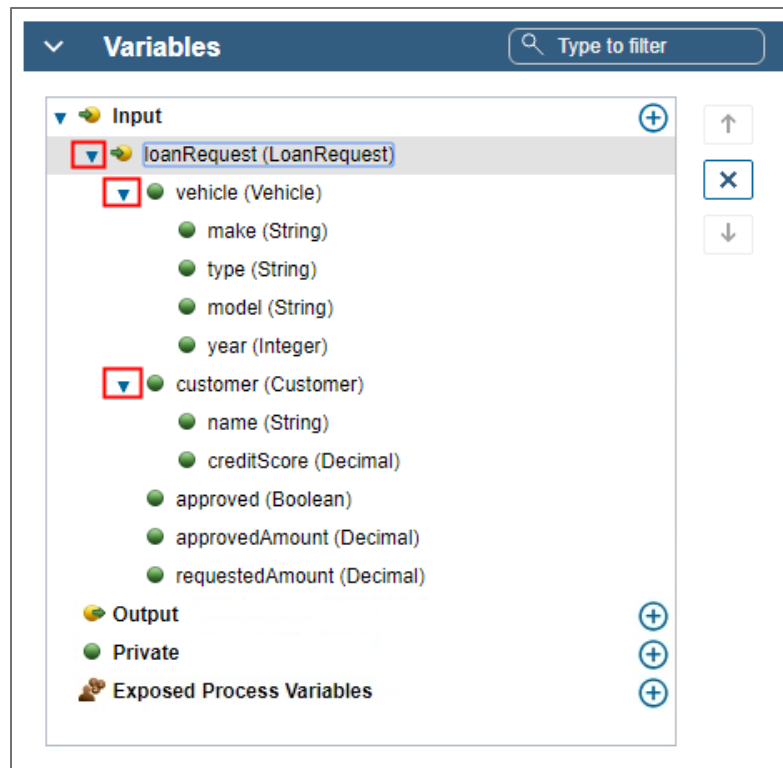
The process consists of two user tasks: **Review Loan Application** and **Approve Loan Request**. The former is executed by a Customer Service team member to review the loan application for completeness before it is sent for approval. The loan request itself is fetched by the **Retrieve Loan Request** system task from a backend system (in this example this is simulated with a script). The **Approve Loan Request** task is completed by the Underwriter team who is responsible for approving or rejecting the application based on the information provided. Finally, the approval result is emitted to IBM Business Automation Insights through the **Track Approval Result** tracking event.

Let's now look at the data.

- \_\_\_ 8. Click the **Variables** tab in the process editor



- \_\_\_ 9. Expand the **loanRequest** input variable, to see the data structure of a loan request, by clicking on the red boxes illustrated below



Three business objects have been created to support this process: **LoanRequest**, **Vehicle** and **Customer**. The main business object is **LoanRequest** and it references the **Customer** and **Vehicle** business objects.

Note that this example is not intended to reflect a real loan approval system, which is notably more complex. The loan request contains information on the vehicle to be financed (its **make**, **type**, **model**, and **year**) and information about the customer, in particular the **creditScore** property, which represents the customer's overall credit health. It also contains information about the loan request itself such as the requested amount, the approval status and the amount to be lent.

Since this is not a real process, the **loanRequest** variable is initialized with random values by the **Retrieve Loan Request** system task.

Let's now look at the tracking of this data in **IBM Business Automation Insights**.

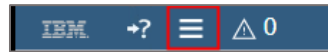
\_\_\_ 10. Return to the process flow by clicking on the **Definition** tab



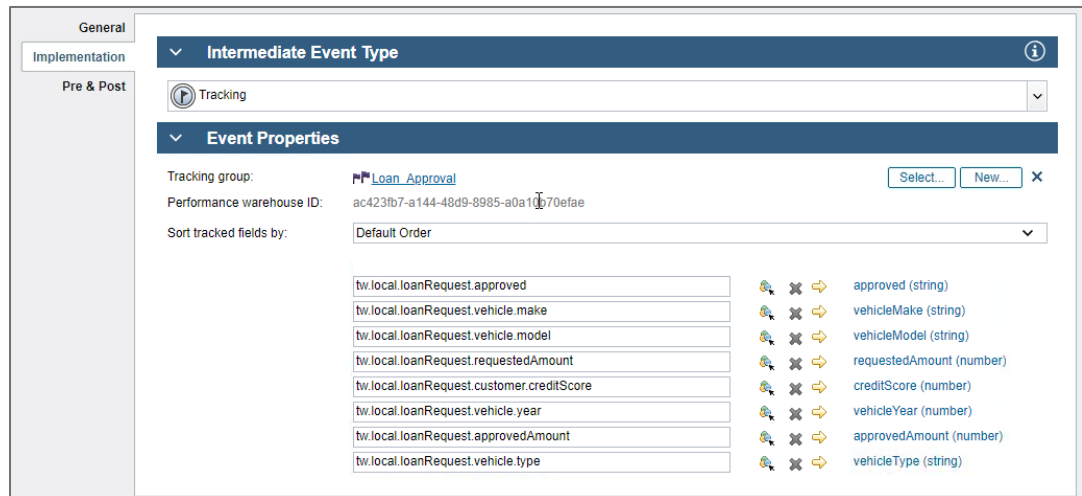
\_\_\_ 11. Select the Tracking Approval Result tracking event



- \_\_\_ 12. If the properties pane is not visible, click on the **Properties** button on the bottom banner of the designer



- \_\_\_ 13. Click the **Implementation** tab



The screenshot shows the 'Implementation' tab for an 'Intermediate Event Type'. The 'Event Properties' section is expanded, displaying the following configuration:

- Tracking group: **Loan\_Approval** (with 'Select...' and 'New...' buttons)
- Performance warehouse ID: `ac423fb7-a144-48d9-8985-a0a10b70efae`
- Sort tracked fields by: **Default Order**

Tracked Field	Process Variable	Data Type
<code>tw.local.loanRequest.approved</code>	<code>approved</code>	string
<code>tw.local.loanRequest.vehicle.make</code>	<code>vehicleMake</code>	string
<code>tw.local.loanRequest.vehicle.model</code>	<code>vehicleModel</code>	string
<code>tw.local.loanRequest.requestedAmount</code>	<code>requestedAmount</code>	number
<code>tw.local.loanRequest.customer.creditScore</code>	<code>creditScore</code>	number
<code>tw.local.loanRequest.vehicle.year</code>	<code>vehicleYear</code>	number
<code>tw.local.loanRequest.approvedAmount</code>	<code>approvedAmount</code>	number
<code>tw.local.loanRequest.vehicle.type</code>	<code>vehicleType</code>	string

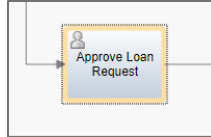
The main task in this process is to approve or reject a loan application and determine the amount of the loan. Once the approval decision is taken and the task completed, the result is stored in **IBM Business Automation Insights** by emitting a tracking event with the relevant information. Tracking events reference a tracking group that defines the data to be tracked. The **Loan\_Approval** tracking group was defined for this purpose. It defines the same set of fields as our loan request, including the field that determines if the loan has been approved.

In the configuration of the tracking event above we map the fields of the process variable **loanRequest** into the corresponding tracking group fields. Once the process engine executes the node, it will emit an event with the corresponding data which will be stored in the BAI data lake. You will use this data to train a machine-learning model that provides an approval recommendation. The name of the tracking group, **Loan\_Approval**, will be used to find the relevant data in BAI.

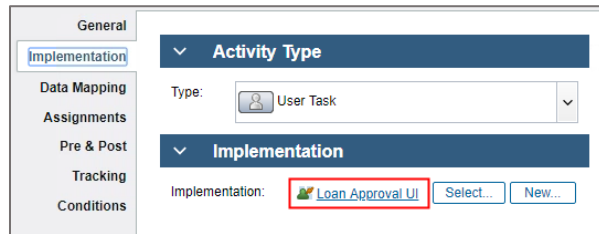
### 1.3 Examine the Loan Approval Task UI

Let's look at the **Approve Loan Request** task implementation. You will later modify this client side human service to call a machine-learning model to provide a recommendation on whether to approve a loan request. You will modify the user interface to display the recommendation.

- \_\_\_ 14. Select the **Approve Loan Request** task in the process flow



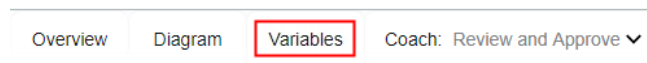
- \_\_\_ 15. Select the **Implementation** tab in the **Properties** pane
- \_\_\_ 16. Click on the **Loan Approval UI** link to open the client side human service used to implement this task



The Loan Approval UI client side human service is very simple. It contains a single coach, **Review and Approve**, as shown below. The coach implements the user interface for the underwriter to review the loan application and approve or reject the request.

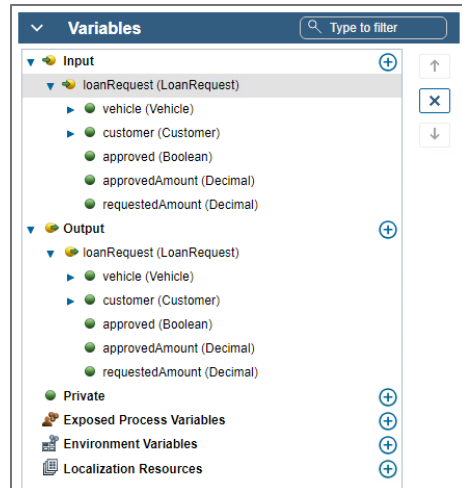


- \_\_\_ 17. Click on the **Variables** tab in the client side human service editor



As shown below, the service receives and returns a **LoanRequest** business object. This variable is used by the coach and modified with the approval information.





\_\_\_ 18. Click on the **Coach** tab

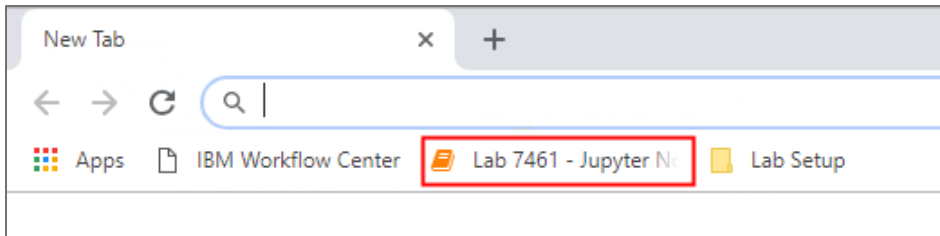


The coach defines the UI used by the underwriter. It contains sections for the client's information, vehicle and loan details. At the bottom of the form there are also widgets to allow the underwriter to approve the request and enter the amount approved.

\_\_\_ 19. Close all editor and continue with the next section

## 2 Train and Deploy a Machine Learning Model to provide Loan Approval Recommendation

To complete this section, you will work with the Jupyter notebook provided (**7461.ipynb**). Launch the notebook by opening the Chrome browser and clicking on the corresponding **Lab 7461 – Jupyter Notebook** in the bookmark bar.



Follow the instructions in the notebook and, once completed, return to this document to continue with [Section 3](#).

A version of the notebook source is included in [Appendix A](#).

## 3 Add Recommendation to the Car Loan Approval Process

In this section you will once again work in **IBM Business Automation Workflow**. You will modify the **Loan Approval UI** client side human service to display the output of the machine-learning model you built in [Section 2](#) and provide a recommendation to approve or deny a loan application.

### 3.1 Implement the Watson Scoring API Call

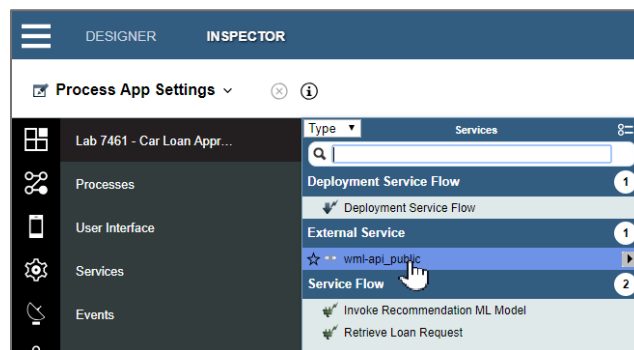
**IBM Business Automation Workflow** can integrate with the Watson Machine Learning service through its REST API. You will use a Service Flow (**Invoke Recommendation ML Model**) to make the REST call to the Watson scoring API through a REST External Service (**wml-api\_public**). These two artifacts are provided for you but require updates to invoke your machine learning model.

#### 3.1.1 Examine the External Service

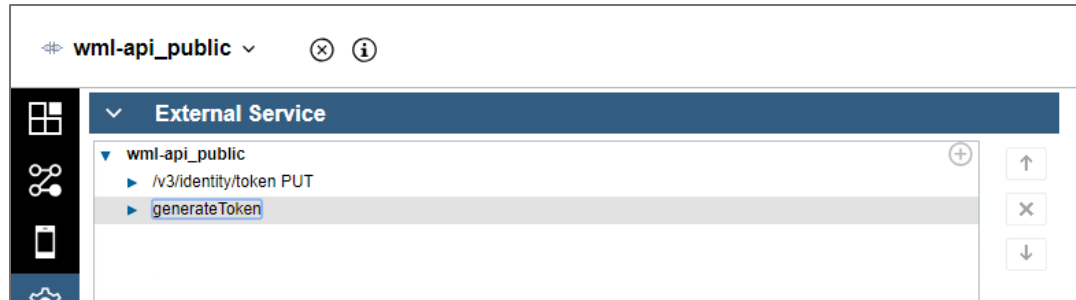
External services provide a means to integrate with an external system. These can be discovered from a WSDL, Swagger or Java class. For this lab the **wml-api\_public** external service has been discovered from the **IBM Watson Machine Learning API** swagger (at [https://watson-ml-api.mybluemix.net/swagger/wml-api\\_public.json](https://watson-ml-api.mybluemix.net/swagger/wml-api_public.json)). Its purpose is to enable the application to call the scoring API for the model that you deployed in the previous section.

Let's look at the external service.

- \_\_\_ 1. Open the **wml-api\_public** external service: **Services** category > **wml-api\_public** external service

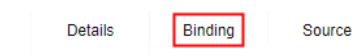


- \_\_\_ 2. Examine the service interface



The operations that are displayed in the tree are invocable from a **Service Task** in a **Service Flow**. For REST external services, such as this one, there may be additional operations that are only invocable through the JavaScript API. Other operations are only invocable through the Java Script API. These are highlighted during discovery and can also be seen the **Source** tab on the right panel. For this lab, you will use the ***generateToken*** operation as well as the scoring operation invoked through JavaScript.

- \_\_\_ 3. Select the root tree element on the tree (***wml-api\_public***) and then click on the **Binding** tab in the right panel

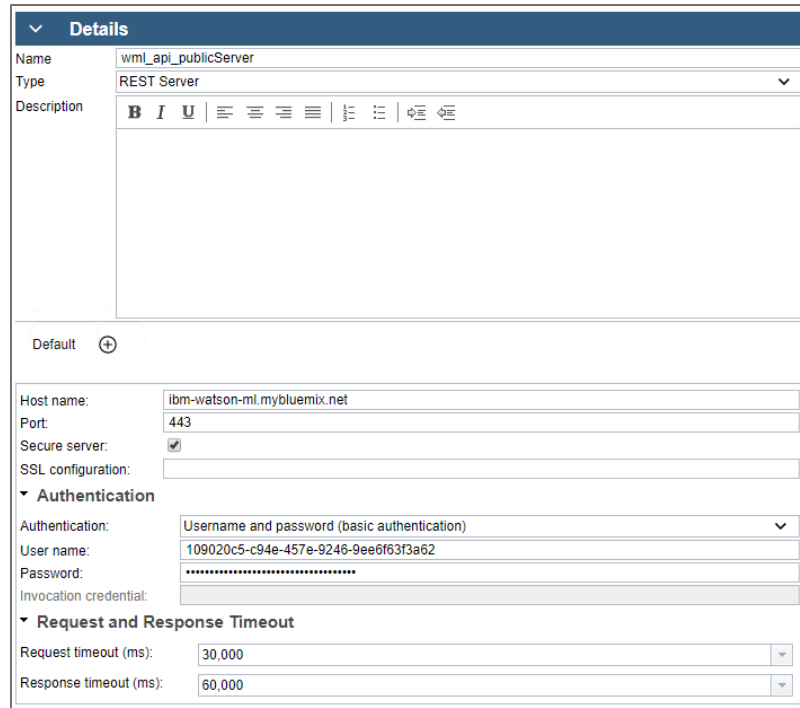


- \_\_\_ 4. Examine the binding details



The service uses a REST binding. It references a REST server, **wml\_api\_publicServer**, which is defined in the **Process App Settings**.







- \_\_\_ 5. Click the **wml\_api\_publicServer** link to open the server definition




**Details**

Name: wml\_api\_publicServer

Type: REST Server

Description: **B I U** |  |  |  |  |  | 

Default 

Host name: ibm-watson-ml.mybluemix.net

Port: 443

Secure server: ☒

SSL configuration:

**Authentication**

Authentication: Username and password (basic authentication)

User name: 109020c5-c94e-457e-9246-9ee6f63f3a62

Password:

Invocation credential:

**Request and Response Timeout**

Request timeout (ms): 30,000

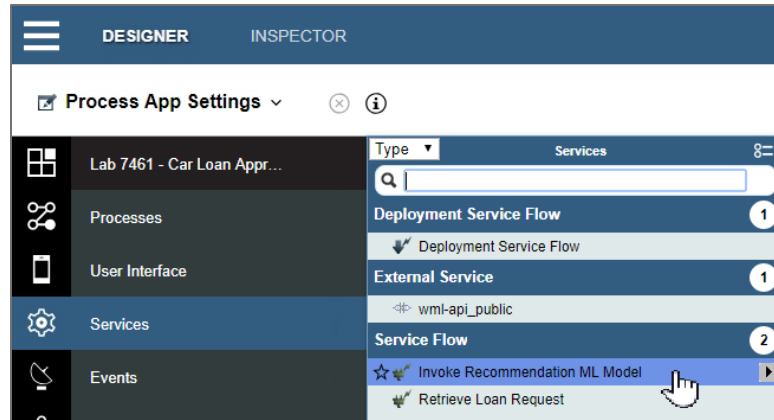
Response timeout (ms): 60,000

The server editor opens showing the details. The server definition captures the endpoint information for the REST API. There are no changes required here as the server is set up for the **Watson Machine Learning** service configured for this lab. If you use your own service you need to update the following information: host name, user name and password. You can obtain these values from the credentials associated with your **Watson Machine Learning** service. See [here](#).

### 3.1.2 Update the Invoke Recommendation ML Model Service Flow

This **Invoke Recommendation ML Model** service flow implements the call to the machine learning model deployed in Section 2. You will modify it with the deployment and model IDs that you recorded in Section 2.

- \_\_\_ 6. Open the **Invoke Recommendation ML Model** service flow: **Services** category > **Invoke Recommendation ML Model** service flow

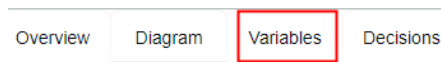


\_\_\_ 7. Examine the service

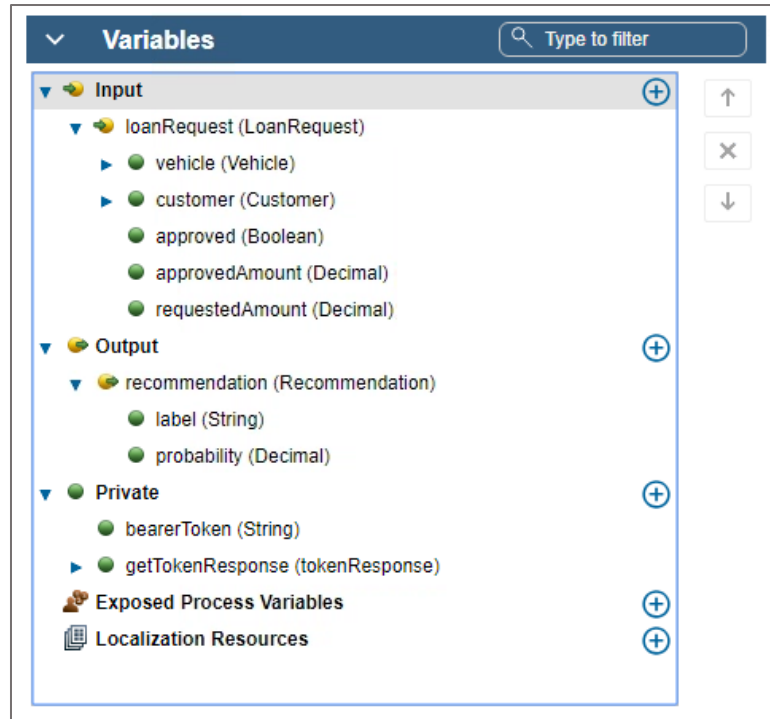


The service has two nodes. **Get Token** retrieves a token used to authenticate the second REST call executed by the **Get Recommendation from Watson** node.

\_\_\_ 8. Switch to the **Variables** tab

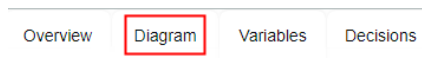


\_\_\_ 9. Examine the variable definitions

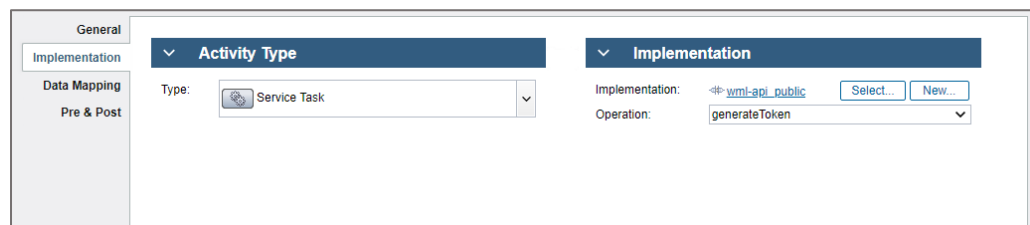


The service takes a **LoanRequest** object as input and return a **Recommendation** object as output. The result from the **Get Recommendation from Watson** task will be stored into the **recommendation** variable. The **label** field represents the string value of the result: “true” or “false” corresponding to approve and reject. The **probability** field holds the percentage confidence level that the recommendation is correct.

\_\_\_ 10. Return to the diagram by clicking on the **Diagram** tab

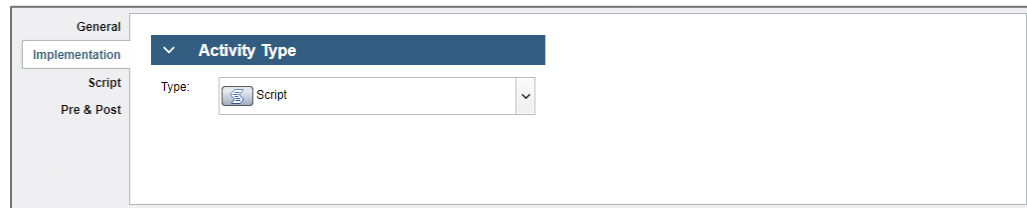


\_\_\_ 11. Select the **Get Token** node and then click the **Implementation** tab in the **Properties** pane below



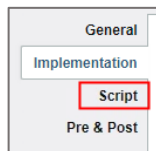
**Get Token** is a **Service Task** node that invokes the **generateToken** operation in the **wml-api\_public** external service.

- \_\_\_ 12. Select the **Get Recommendation from Watson** node then click the **Implementation** tab in the **Properties** pane below

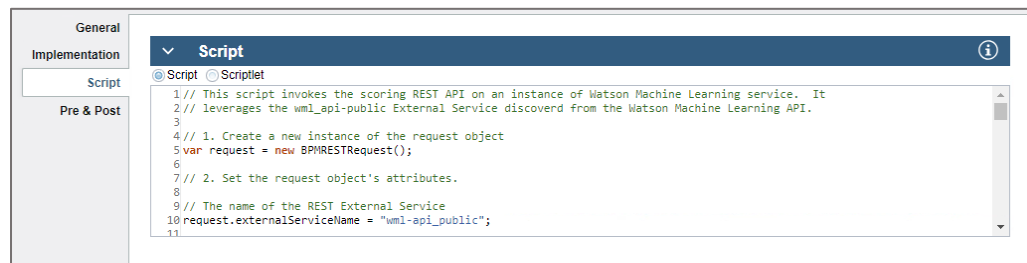


This node also invokes the external service. However, the target operation is not invocable through a **Service Task**, like *generateToken* above. Instead this node is implemented with a **Script**. Also, by using JavaScript to invoke a REST service you have more capabilities and control over setting the input parameters, request headers, authentication information, processing the output parameters, and handling errors.

- \_\_\_ 13. Click on the **Script** tab in the **Property** pane



- \_\_\_ 14. Examine the JavaScript code



The JavaScript call uses the **BPMRESTRequest** object to invoke the target REST API. The scoring operation path is:

*/v3/wml\_instances/{**instance\_id**}/published\_models/{**published\_model\_id**}/deployments/{**deployment\_id**}/online*

You must provide the substitution values bolded above that are specific to your deployed model.

- \_\_\_ 15. Locate where **request.parameters** is set



```

request.parameters = {
  "online_prediction_input" :{
    "fields":["creditScore",
      "requestedAmount",
      "vehicleMake",
      "vehicleModel",
      "vehicleYear",
      "vehicleType"],
    "values":[[tw.local.loanRequest.customer.creditScore,
      tw.local.loanRequest.requestedAmount,
      tw.local.loanRequest.vehicle.make,
      tw.local.loanRequest.vehicle.model,
      tw.local.loanRequest.vehicle.year,
      tw.local.loanRequest.vehicle.type]]
  },
  "instance_id" : "8f794cbe-79cd-4d90-acb2-f797eca3dbec",
  "deployment_id": "YOUR DEPLOYMENT ID",
  "published_model_id" : "YOUR MODEL ID"
};

```

Update the values highlighted above with the corresponding values obtained from deploying your model to Watson Machine Learning. **Note** that if you are using your own service instance you'll need to update the **instance\_id** parameter as well. This ID can be found in your Watson Machine Learning credentials – see [here](#).

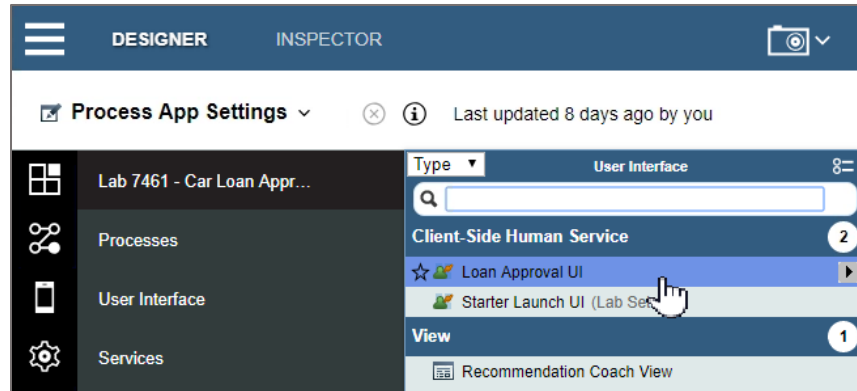
\_\_\_ 16. Finish editing by clicking the button on the top banner



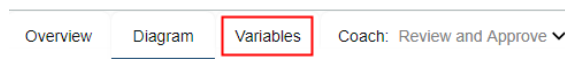
## 3.2 Modify the Approval User Interface to Display the Recommendation

The next step is to update the user interface to show the recommendation that the **Invoke Recommendation ML Model** service will return. You will first modify the service input and then use it with a new coach view.

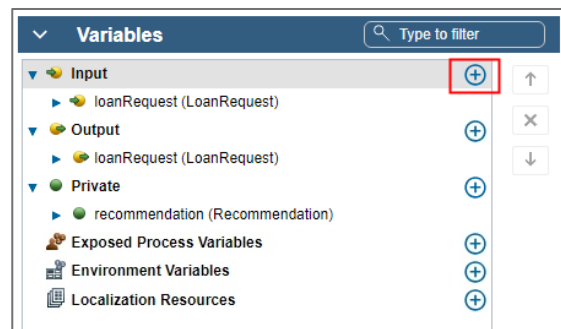
\_\_\_ 17. Open the **Loan Approval UI** client side human service: **User Interface** category > **Loan Approval UI**



\_\_\_ 18. Click on the **Variables** tab

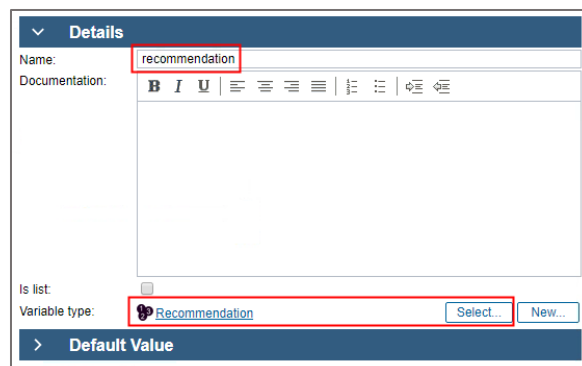


\_\_\_ 19. Click on the '+' button next to the **Input** section to add a new variable



\_\_\_ 20. On the **Details** panel (right) set the name to **recommendation**

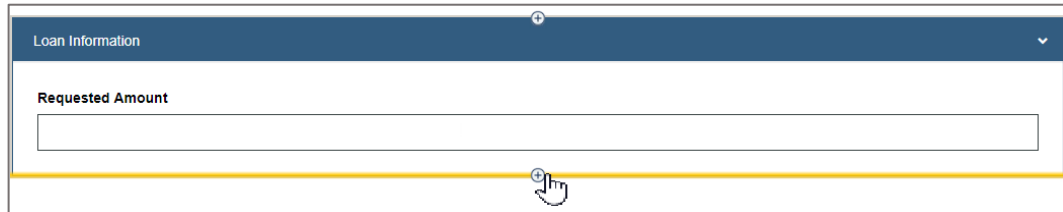
\_\_\_ 21. Set the **Variable type** by clicking the **Select...** button and selecting the **Recommendation** business object



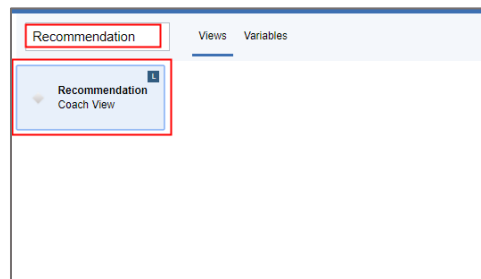
\_\_\_ 22. Switch to the **Coach** tab



- \_\_\_ 23. Scroll down to locate the **Loan Information** panel and click on the '+' that appears upon hovering over the bottom edge

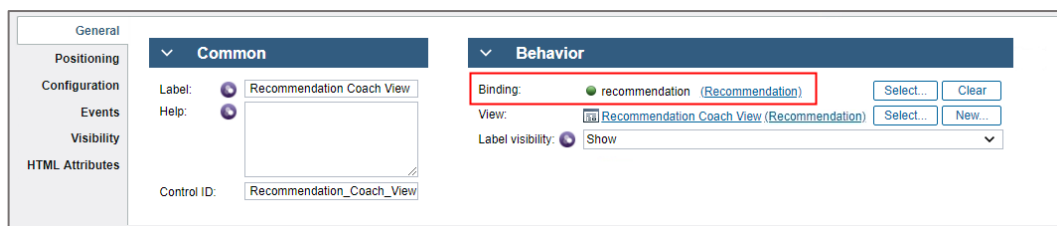


- \_\_\_ 24. The palette opens up. Type '**Recommendation**' on the search box and click on the **Recommendation Coach View** to add it



The **Recommendation Coach View** is provided to you to easily display the recommendation. It is a composite view with two set of widgets: one for approval and one for rejection. The value of the **recommendation** variable controls the visibility for each widget.

- \_\_\_ 25. With the view selected open the **Properties** pane and set the **Binding** to the **recommendation** variable you added earlier. Use the **Select...** button to bring up the variable selector



The changes to the client side human service are complete. Finish editing and proceed to the next section.

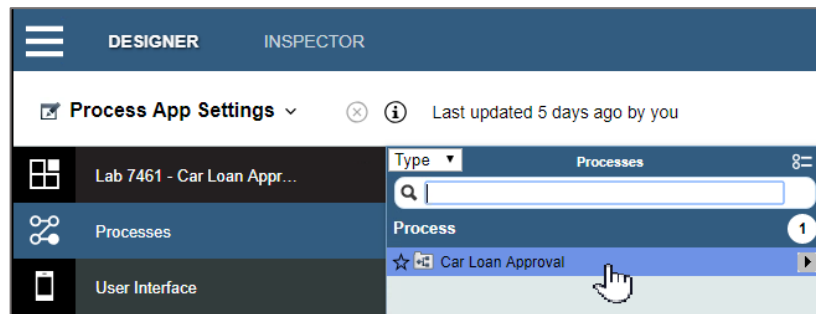
- \_\_\_ 26. Finish editing by clicking the button on the top banner



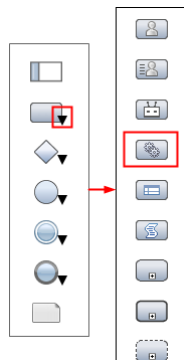
### 3.3 Call the Machine-Learning Model to Obtain a Prediction

You will now modify the **Car Loan Approval** process to get a prediction for a new loan request. The prediction will be passed to the approval task as an input. You will use a **Service Task** to invoke the **Invoke Recommendation ML Model** service flow.

- \_\_\_ 27. Once again, open the **Car Loan Approval** process: **Processes** category > **Car Loan Approval** process

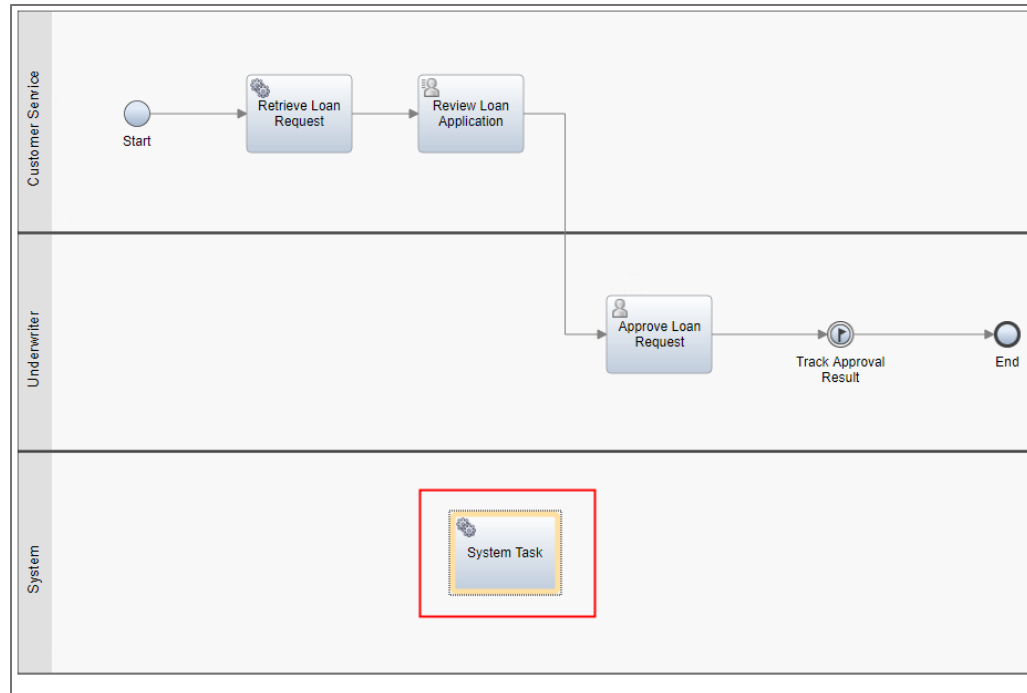


- \_\_\_ 28. Expand the **Activity** drawer on the editor palette by clicking the black arrow on the bottom-right

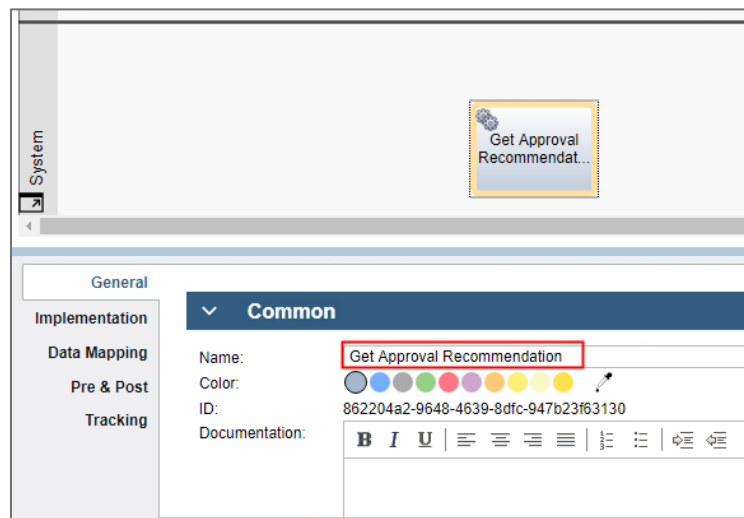


You will use the **System Task** highlighted.

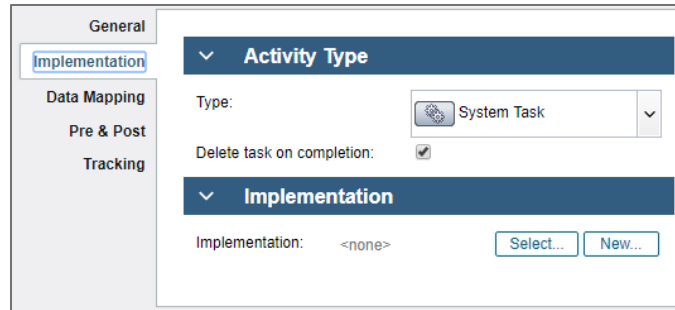
- \_\_\_ 29. Drag and drop a **System Task** from the palette to the **System** lane



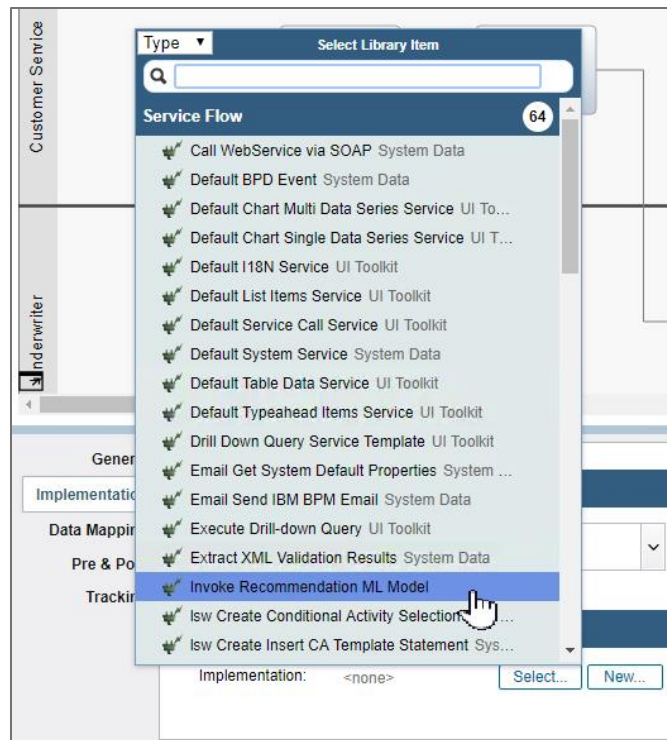
\_\_\_ 30. Update the name of the task to **Get Approval Recommendation** by selecting the node and updating its name in the **General** property tab



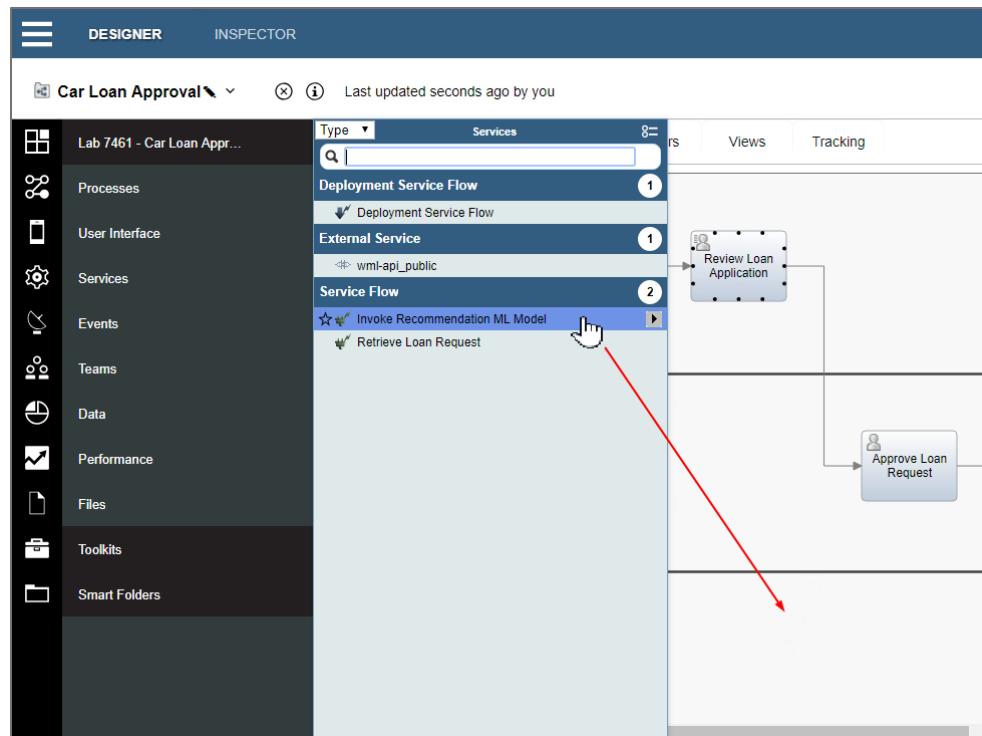
\_\_\_ 31. Select the **Implementation** property tab



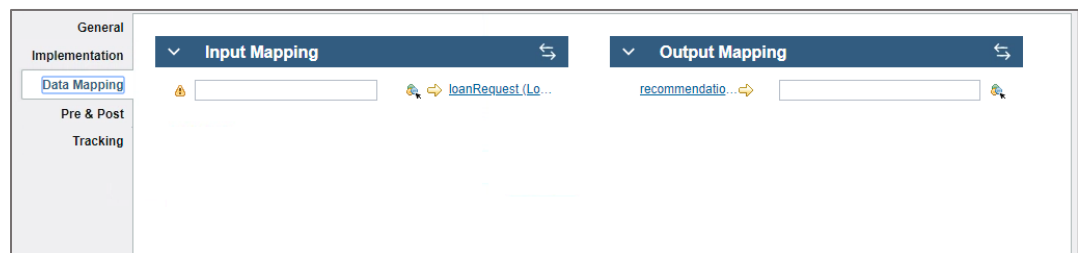
\_\_\_ 32. Click the **Select** button and select the **Invoke Recommendation ML Model** service flow



Now that you understand how a task is built it is worth mentioning an alternative to steps 19-23: drag and drop the **Invoke Recommendation ML Model** service from the **Services** category onto the canvas to create a fully configured **System Task**.

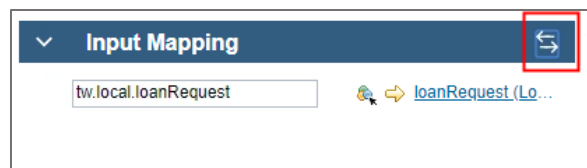


\_\_\_ 33. Click the **Data Mapping** tab



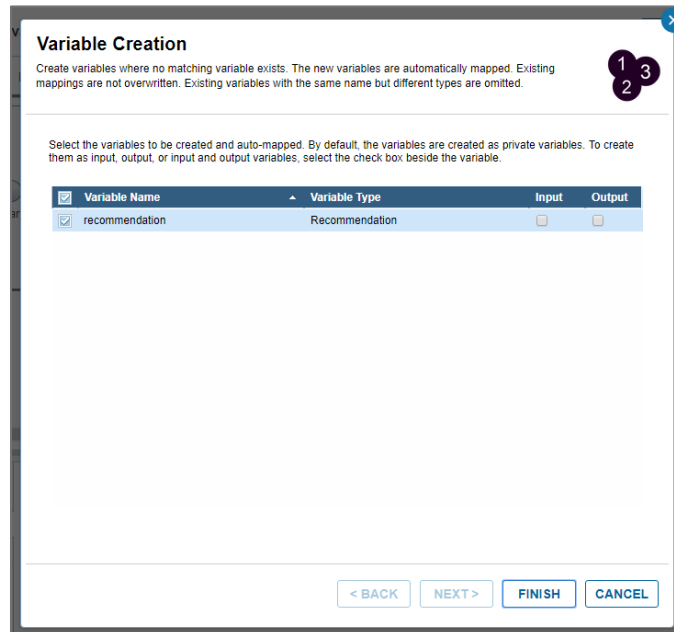
This tab allows you to map local variables into and out of the service. You specify the variable as JavaScript. You can enter it manually, select a variable using the selector to the right of each field, or you can use the auto-mapping feature in the input and output section banners. Let's use auto-mapping

\_\_\_ 34. Click on the auto-mapping button for the **Input Mapping** section



The auto-mapping feature looks at the current set of variables defined in the process. If it finds one that matches by name and type it automatically inserts it into the corresponding mapping field. **tw.local.loanRequest** was a match for the input.

- \_\_\_ 35. Click on the auto-mapping button for the **Output Mapping** section



**Variable Creation**

Create variables where no matching variable exists. The new variables are automatically mapped. Existing mappings are not overwritten. Existing variables with the same name but different types are omitted.

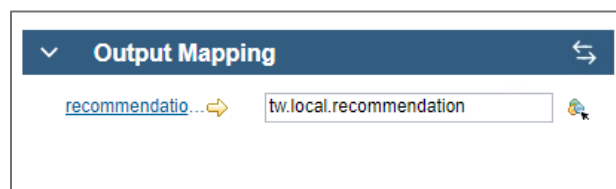
Select the variables to be created and auto-mapped. By default, the variables are created as private variables. To create them as input, output, or input and output variables, select the check box beside the variable.

<input checked="" type="checkbox"/>	Variable Name	Variable Type	Input	Output
<input checked="" type="checkbox"/>	recommendation	Recommendation	<input type="checkbox"/>	<input type="checkbox"/>

< BACK   NEXT >   FINISH   CANCEL

This time the editor did not find a match for the output. As a result, a dialog prompts you to create a new variable with the specification shows in the dialog. You can choose to make this an input and/or output variable by checking the corresponding boxes. If neither is checked, a private variable is created.

- \_\_\_ 36. Click **Finish** to accept the defaults



**Output Mapping**

recommendatio... → tw.local.recommendation

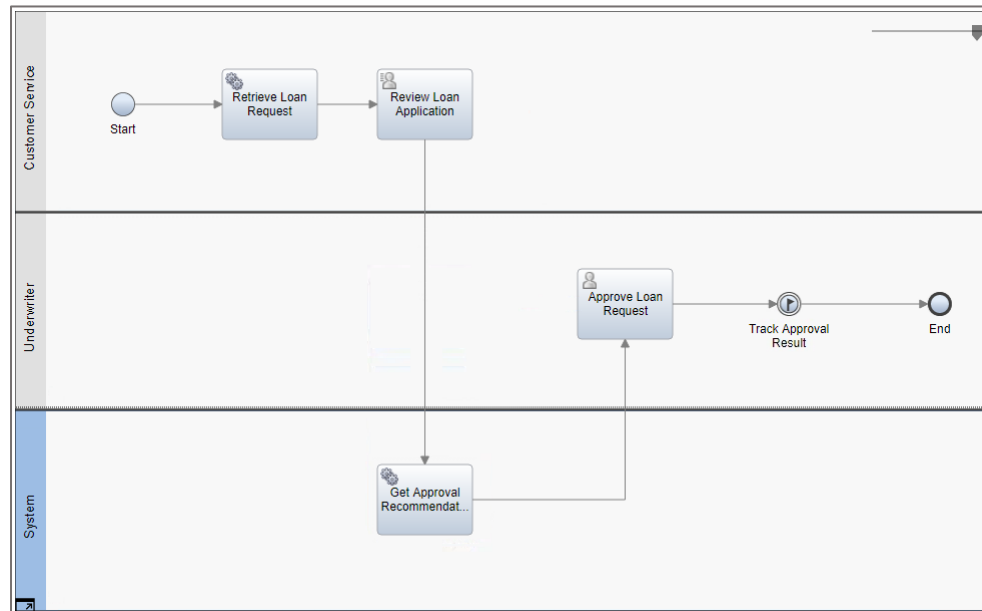
The **recommendation** variable is created and automatically mapped in the corresponding output field.

The **System Task** has been configured. You will now wire it into the flow.

- \_\_\_ 37. Back on the diagram, remove the wire between **Review Loan Application** and **Approve Loan Request** by selecting it and pressing **Delete**

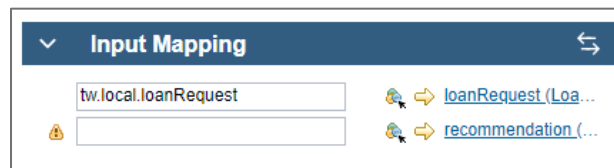


- \_\_\_ 38. Connect the **Review Loan Application** user task to the **Get Approval Recommendation** system task
- \_\_\_ 39. Connect the **Get Approval Recommendation** system task to **Approve Loan Request** user task



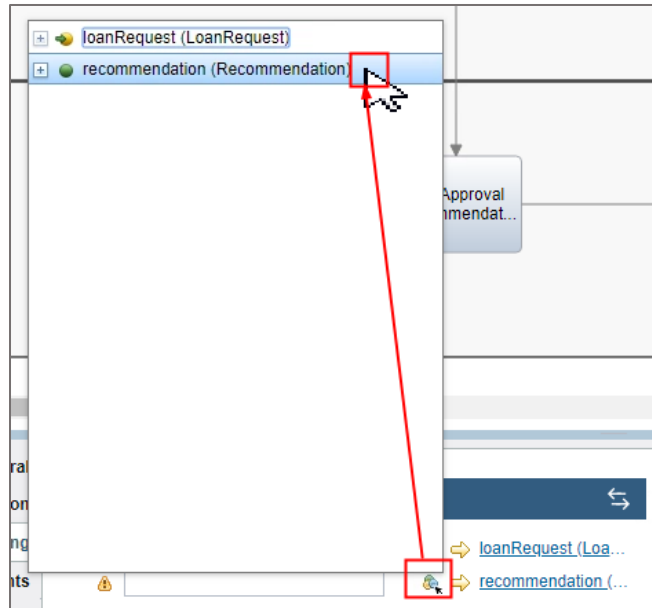
Your diagram should match what is shown above.

- \_\_\_ 40. Select the **Approve Loan Request** user task and click on the **Data Mapping** tab in the **Properties** pane



You added a new input to this service when you modified the user interface. You will pass the data returned by the **Get Approval Recommendation** system task into it.

- \_\_\_ 41. Click the selector next to the field and select the **recommendation** variable



The process changes are complete. You are ready to finish and test your updates.

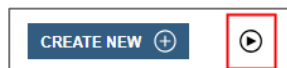
- \_\_\_ 42. Finish editing by clicking the button on the top banner



### 3.4 Test the Application

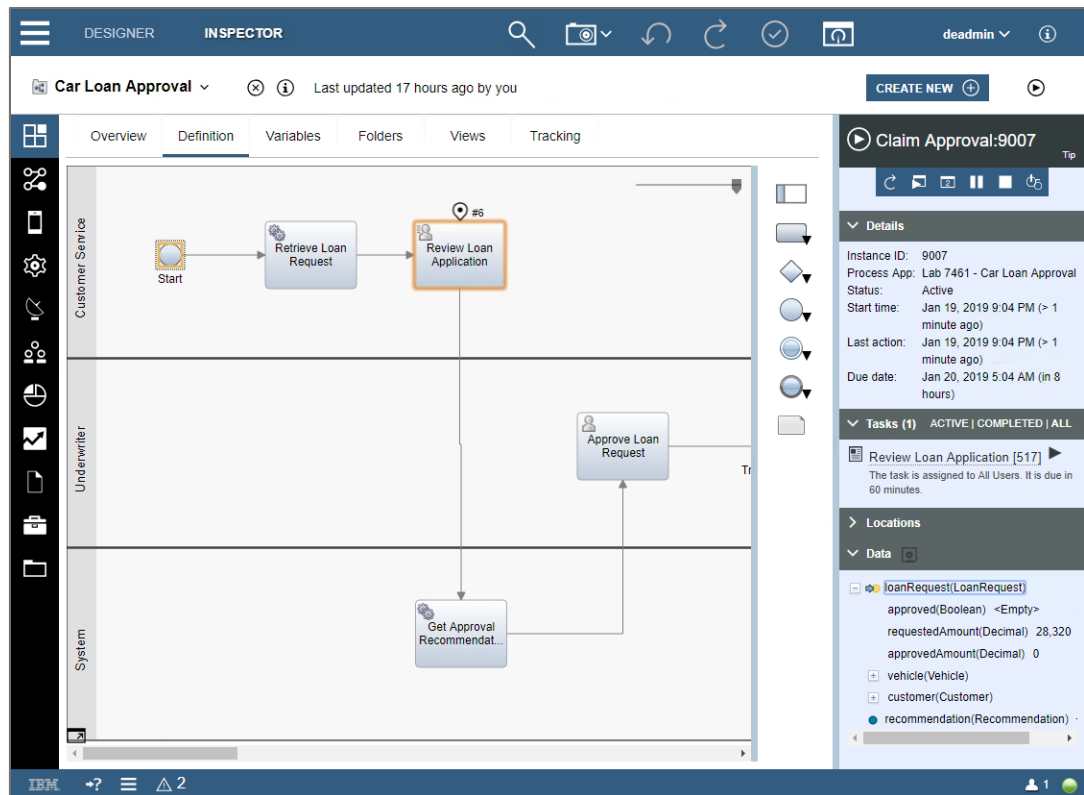
You are ready to test your updates. Use the process inspector to playback the process and see the recommendation in action.

- \_\_\_ 43. Launch a new instance of the **Car Loan Approval** process by clicking on the play button in the top right corner of the editor



The process starts and the **Process Designer** switches to the **Inspector** perspective.

- \_\_\_ 44. Examine the **Process Inspector**

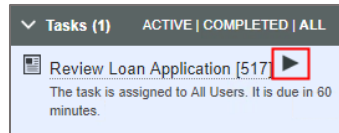


You can interact with the process instance using the panel on the right. The top section contains several actions to interact with the instance. Hover over each one to display their name. Take note of the **Refresh** button (first one on the left) which you can use to update the location of the token if not already at the next user task.

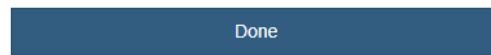


The **Details** section shows general information about the current process instance. The **Tasks** section shows all the tasks in the process and provides a filter to show active, completed or all tasks. Active task can be worked on by clicking on the play button (▶). You can also debug the service by clicking on the debug button (🐛). The debug option is not available for inline user tasks, where the human service is generated. The **Locations** section shows the location of all active tokens in the process instance. These will also be highlighted in the diagram as well (📍). Finally, the **Data** section shows all variables in the process. You can change variable values and save the changes using the save button in the header (💾).

- \_\_\_ 45. Click on the play button next to the **Review Loan Application** task to bring up the task UI and work on it



- \_\_\_ 46. Examine the loan request information received by this process instance. You can modify the data. Complete the task by clicking on the **Done** button at the bottom of the form




Completing this task will trigger the process to continue execution. The process inspect will stop at the next user task in the flow, **Approve Loan Request**.

- \_\_\_ 47. Close the playback browser window to return focus to the Process Designer window
- \_\_\_ 48. Click on the play button next to the **Approve Loan Request** task bring up the task UI and work on it



- \_\_\_ 49. Examine the approval form

Loan Applicant Information		
<b>Customer Name</b>		
John Smith		
<b>Credit Score</b>		
65		
Financed Vehicle Information		
<b>Type</b>		
Car		
<b>Make</b>	<b>Model</b>	<b>Year</b>
Ford	Focus	1,997
Loan Information		
<b>Requested Amount</b>		
8,047		
Loan Approval Recommendation		
 <p>Based on previous decisions we recommend to <b>APPROVE</b> the loan with a confidence of 92</p>		
<input type="checkbox"/> I Approve the loan		
<b>Approved Amount</b>		
0		
<div>Complete Approval</div>		

The form shows the result of the recommendation in the new section that you added. At this point you can accept the recommendation or rejected it. The approval is confirmed using the **I Approve the loan** checkbox and **Approved Amount** field.

\_\_\_ 50. Complete the process by clicking on the **Complete Approval** button.

You are encouraged to playback the process several times to experiment with different values and see the result of the recommendation.

## Summary

Congratulations! you've optimized a process using machine-learning to improve decision making.

In the first part of this lab you used **IBM Business Automation Workflow** Process Designer to inspect and understand how an existing application works.

In the second part you then used a Jupyter notebook to extract data from **IBM Business Automation Insights** and used it to train and deploy a machine learning model that can provide a recommendation.

In the third part you modified and tested the application to leverage this model and help the user make a final decision on the loan approval.

Finally, you used the Inspector perspective in the Process Designer to test your updates.

## **We Value Your Feedback!**

- Don't forget to submit your Think 2019 session and speaker feedback! Your feedback is very important to us – we use it to continually improve the conference.
- Access the Think 2019 agenda tool to quickly submit your surveys from your smartphone, laptop or conference kiosk.

## Appendix A: Jupyter Notebook

### 4 Lab Center – Hands-on Lab

#### 4.1 Session 7461

#### 4.2 Session Title **Add Intelligence to Business Automation with IBM Business Automation Insights**

**Christophe Jolif**, IBM Digital Business Automation, Architecture & Development,  
christophe.jolif@fr.ibm.com

**Sebastian Carbajales**, IBM Digital Business Workflow, Architecture & Development,  
sebastia@ca.ibm.com

You have completed **Section 1** in the lab by inspecting the as-is process. You are now ready to work with the historical data generated by the process and create a machine-learning model that will provide a recommendation to approve or reject a loan request.

**You will use this Python Jupyter notebook to accomplish this goal.**

#### 4.3 Jupyter Notebook Introduction

A Jupyter notebook is a web-based environment for interactive computing. You can run small pieces of code that process your data, and you can immediately view the results of your computation.

Notebooks include all of the building blocks you need to work with data:

- Loading of the data
- The code computations that process the data
- Visualizations of the results
- Text and rich media to enhance understanding

Code computations can build upon each other to quickly unlock key insights from your data. Notebooks record how you worked with data, so you can understand exactly what was done, reproduce computations reliably, and share your findings with others.

##### 4.3.1 The cells in a Jupyter notebook

A Jupyter notebook consists of a sequence of cells. The flow of a notebook is sequential. You enter code into an input cell, and when you run the cell, the notebook executes the code and prints the output of the computation to an output cell.

You can change the code in an input cell and re-run the cell as often as you like. In this way, the notebook follows a read-evaluate-print loop paradigm.



### 4.3.2 Useful Shortcuts

Use the following shortcuts to execute the code in a cell:

- **Run cell:** CTRL + ENTER
- **Run cell, select below:** SHIFT + ENTER

For a full list of shortcuts: *Help > Keyboard Shortcuts*

### 4.3.3 Do Want to Discover More?

Take a look of the Jupyter notebook interface. Launch it from *Help > User Interface Tour*.

## 4.4 Train and Deploy a Machine Learning Model to provide Loan Approval Recommendation

You will now execute the code in this notebook to train and deploy a machine learning model. You will integrate this model, in Section 3 of the lab, with the Car Loan Approval process to provide a recommendation to approve or reject a loan request.

You will perform the following steps:

1. [Load the time series data for the As-Is process](#)
2. [Explore the format of the data and interpret it](#)
3. [Create an Apache® Spark machine learning model](#)
4. [Store the model in Watson ML](#)
5. [Deploy a model](#)
6. [Test the deployed model](#)

The code in this notebook is ready to execute, but you are encouraged to experiment with it by changing it and re-executing cells to see the effect of your change. Should you need to undo your changes you can revert to a previous checkpoint using the `**File** > **Revert to checkpoint**` menu.

The following cell contains all the dependencies required to run this notebook. The code can be uncommented and executed for a new environment.

In [ ]:

```
## Install PySpark
# !rm -rf $PIP_BUILD/pyspark
# !pip install --upgrade pyspark==2.1.3

## Install visualization packages
# !pip install --upgrade matplotlib
# !pip install --upgrade seaborn
```

```
## Install Numpy
# !pip install numpy

## Install the Watson Machine learning API Package
# !rm -rf $PIP_BUILD/watson-machine-learning-client
# !pip install --upgrade watson-machine-learning-client==1.0.260
```

## 4.5 Step 1: Load the time series data for the As-Is process

### 4.5.1 The format of the IBM Business Automation Insights data

Events emitted while a process executes are stored in IBM Business Automation Insights. Several event types are supported but in this scenario you only need the events that are recorded when a tracking point executes. These are stored as bpm-timeseries for tracking data. Every time a process executes a tracking point, a record is added to HDFS in the form of JSON data.

In this scenario, the timeseries data is partitioned by the following elements:

- The identifier and version number of the Workflow business process application
- The tracking group identifier

Thus, HDFS file names start with the following path:

*[hdfs root]/ibm-bai/bpmn-timeseries/[processAppId]/[processAppVersionId]/tracking/[trackingGroupId]*

Remember, the tracking group name is Loan\_Approval. To find the data, you query the various IDs from the Workflow system.

*Refer*

to [https://www.ibm.com/support/knowledgecenter/en/SSYHZ8\\_18.0.x/com.ibm.dba.bai/topics/ref\\_bai\\_data\\_paths.html](https://www.ibm.com/support/knowledgecenter/en/SSYHZ8_18.0.x/com.ibm.dba.bai/topics/ref_bai_data_paths.html) for more details on HDFS data paths.

[Demo - Next Visualize](#)

### 4.5.2 Finding the application ID and version, and the tracking group ID

You will use the IBM Business Automation Workflow REST API to retrieve the application and tracking group information. You will then use these values to build the HDFS path described in the previous section.

*You can refer*

to [https://www.ibm.com/support/knowledgecenter/en/SSYHZ8\\_18.0.x/com.ibm.dba.bai/topics/tsk\\_bai\\_retrieve\\_bpmn\\_id.html](https://www.ibm.com/support/knowledgecenter/en/SSYHZ8_18.0.x/com.ibm.dba.bai/topics/tsk_bai_retrieve_bpmn_id.html) for more information on how to retrieve BPMN identifiers.

The Python code below sets up the REST API URL.

In [ ]:

```
import urllib3, requests, json
urllib3.disable_warnings()
```

```
bpmusername='tw_admin'
bpmpassword='tw_admin'
bpmrestapiurl = 'https://localhost:9443/rest/bpm/wle/v1'
```

```
headers = urllib3.util.make_headers(basic_auth='{username}:{password}'.format(
    username=bpmusername, password=bpmpassword, verify=False))
```

You now retrieve the *process application ID and version number* by using the processApps REST API. The code below searches for the 'Lab 7461 - Car Loan Approval' application and assumes that only one version or snapshot is installed.

In [ ]:

```
url = bpmrestapiurl + '/processApps'
response = requests.get(url, headers=headers, verify=False)

[processApp] = [x for x in json.loads(response.text).get('data').get('process
AppList') if x.get('name') == 'Lab 7461 - Car Loan Approval']

processAppId = processApp.get('ID')

# Note that the first 5 characters of the process app ID below are removed.
All BPM artifact IDs are prefixed
# with the artifact type. In this case, '2066.' indicates this is a process
app ID.
# The REST API returns the full process application id, including its prefix.

print("Process application ID: " + processAppId[5:])

# Get the first snapshot - assume only one - this is the app version ID
snapshot = processApp.get('installedSnapshots')[0]
processAppVersionId = snapshot.get('ID')
print("Process application version ID: " + processAppVersionId)
```

Next you retrieve the *tracking group ID* using the 'assets' REST API. You specify the process app ID, just computed, and asset type to filter the results to tracking groups defined within the 'Lab 7461 - Car Loan Approval' application. You then retrieve the 'Loan\_Approval' tracking group from the results.

In [ ]:

```
url = bpmrestapiurl + '/assets'

response = requests.get(url, headers=headers, verify=False, params={'processA
ppId': processAppId, 'filter': 'type=TrackingGroup' })
```

```
[trackingGroupId] = [x.get('poId') for x in json.loads(response.text).get('data').get('TrackingGroup') if x.get('name') == 'Loan_Approval']
```

```
# Note that the first 3 characters of the tracking group ID below are removed
. As in the process app case
# this is the prefix to indicate this is a tracking group ID.
```

```
print('Tracking group ID : ' + trackingGroupId[3:])
```

All required information to build the HDFS path has been obtained. You now continue to query the data.

### 4.5.3 Using Spark SQL to read IBM Business Automation Insights data

IBM Business Automation Insights stores data in HDFS. As described above, the events coming from the Workflow instance are stored in JSON files.

The code below is already configured with the target HDFS URL for the lab environment.

In [ ]:

```
from pyspark.sql import SparkSession
```

```
hdfs_root = 'hdfs://hdfs1.ibm.edu/think2019'
```

```
spark = SparkSession.builder.getOrCreate()
```

```
spark.conf.set("dfs.client.use.datanode.hostname", "true")
```

```
# Get the timeseries Dataset by reading the JSON data from HDFS
```

```
timeseries = spark.read.json(hdfs_root + "/ibm-bai/bpmn-timeseries/" + processAppId[5:] + '/' + processAppVersionId + '/tracking/' + trackingGroupId[3:] + '/*/*')
```

```
# If BAI were not available but you have the data in a local file you can load it this way instead.
```

```
# timeseries = spark.read.json("sample_loan_approval.json")
```

Note that the various ids for the path are specified in the JSON path. This HDFS path could also use HDFS wildcards. Here, the \* character replaces any directory or file name in the path.

The data is loaded, let's take a quick look. The code below will show a sample of the data, the schema and the number of records available.

In [ ]:

```
# Displays the top 20 rows of Dataset in a tabular form.
```

```
timeseries.show()
```

```
# Print the schema in a tree format
```

```
timeseries.printSchema()
```

```
# Finally, print the total number of events
```

```
print ('The data contains ' + str(timeseries.count()) + ' events')
```

For this lab, we're interested in the data that was tracked by the process. Looking at the schema, this data is contained within the 'trackedFields' attribute of each event.

The code below creates a temporary view on the data so that we can select just the tracked fields from the events.

In [ ]:

```
# Creates a local temporary view called 'timeseries'
```

```
timeseries.createOrReplaceTempView("timeseries")
```

```
# Select all tracked fields
```

```
businessdata = spark.sql("SELECT trackedFields.* from timeseries")
```

```
# Displays the top 20 rows of Dataset in a tabular form.
```

```
businessdata.show()
```

```
# Print the schema in a tree format
```

```
businessdata.printSchema()
```

Let's clean up the column names by removing the type suffix from each column.

In [ ]:

```
businessdata = businessdata.withColumnRenamed("approved.string", "approved")
```

```
businessdata = businessdata.withColumnRenamed("creditScore.integer", "creditScore")
```

```
businessdata = businessdata.withColumnRenamed("requestedAmount.integer", "requestedAmount")
```

```
businessdata = businessdata.withColumnRenamed("approvedAmount.integer", "approvedAmount")
```

```
businessdata = businessdata.withColumnRenamed("vehicleMake.string", "vehicleMake")
```

```
businessdata = businessdata.withColumnRenamed("vehicleModel.string", "vehicleModel")
```

```
businessdata = businessdata.withColumnRenamed("vehicleType.string", "vehicleType")
```

```
businessdata = businessdata.withColumnRenamed("vehicleYear.integer", "vehicleYear")
```

```
businessdata.printSchema()
```

## 4.6 Step 2: Explore the format of the data and interpret it

In this step you will learn a few techniques to understand the data that you are working with and determine what characteristics may be relevant to your prediction. In this exercise you want to be able to predict whether to approve or reject a loan request. So you will look at the

relationships between the 'approved' field and the rest of them to determine which ones may be a good predictor.

### 4.6.1 Data manipulation libraries

In the previous section you used Spark to load the data. You will use the result to also train the Spark model. However, in this section we convert the Spark dataframe to Pandas. This is another table manipulation library but it plays well with visualization libraries. Spark, on the other hand, does not. This lab uses the Seaborn visualization library.

You can find more information on Pandas here <http://pandas.pydata.org/pandas-docs/stable/>.

The Python code below creates a Pandas DataFrame from our Spark data.

In [ ]:

```
import pandas as pd
```

```
summary_pd = businessdata.toPandas()
```

Now we can take a quick look at the DataFrame.

In [ ]:

```
# This prints just the columns in the data frame
```

```
print(summary_pd.columns)
```

```
# Use the head() function to preview the first n rows in the data frame. If  
# you don't pass
```

```
# a value to the function, the default is 5.
```

```
summary_pd.head()
```

In [ ]:

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
```

```
%matplotlib inline
```

#### 4.6.1.1 Using Box Plots to see relationship between categorical and numerical variables

In this lab we want to predict the 'approved' field, so we want to understand the relationship between this field and the others to determine which will influence the value of 'approved'. Our target field is a categorical variable because it can only contain a value of true or false (1 or 0), as opposed to say, 'creditScore' that can contain any value within a range. 'creditScore' is a numerical variable.

A good way to visualize categorical variables is by using boxplots. Let's first examine the relationship between the 'approved' and 'vehicleYear' fields. The boxplot below shows this relationship.

In [ ]:

```
sns.boxplot(x="approved", y="requestedAmount", data=summary_pd)
```

#### 4.6.1.2 Using Categorical Plots to see the breakdown in the distributions

Categorical plots can be used to break down the distributions in a box plot into additional categorical variables. The code below plots the approved/rejected distributions, for each vehicle make, against the vehicle model year.

In [ ]:

```
sns.catplot(x="vehicleMake", y="requestedAmount", hue="approved", data=summary_pd, kind="box")
```

We can see from the plot that the distribution for each make is different. This would imply that vehicle make itself also has influence on the approval recommendation.

Try changing the x and y inputs to the plot to see the relationships between other fields. Use categorical fields for x (vehicleMake, vehicleType) and use numerical fields for y (vehicleYear, requestedAmount, creditScore).

#### 4.6.2 Data Characteristics

We can take a look at the statistics of your data by using the `DataFrame.describe()` function. It will compute basic statistics for all variables. If invoked with no arguments, it will analyze only continuous variables.

In [ ]:

```
sns.countplot(x="vehicleMake", hue="approved", data=summary_pd)
```

[Demo - Next Train](#)

Try changing x to the other categorical variables: *vehicleType* and *vehicleModel*.

#### 4.6.3 What have we observed?

The techniques above gave us a picture of the data we are working with. Using the box plots reveal that vehicleYear, creditScore and requestedAmount, all numerical variables, can potentially be good predictors of approved. The distribution of records between approved and rejected are different enough.

Using categorical plots to look at further breakdown of the categories in these groups also reveal that the distributions are different enough that the categorical variables vehicleMake, vehicleModel and vehicleType, are also potentially good predictors.

Using the data characteristic analysis we can also confirm that there are enough samples in the data. The value ranges in the numerical variables are wide, and the value counts on the categorical variables show that all possible categories are well represented.

### 4.7 Step 3: Create an Apache® Spark machine-learning model

IBM Watson Machine learning supports a growing number of IBM or open-source machine-learning and deep-learning packages. This example uses Spark ML and, in particular, the

Random Forest Classifier algorithm. In this section you will prepare the data, create an Apache® Spark machine-learning pipeline, and train the model.

The first step is to import the libraries required.

In [ ]:

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer, IndexToString, VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml import Pipeline, Model
```

### 4.7.1 Adaptation of data

In this section you will combine multiple complex algorithms into a single pipeline. These algorithms will be applied to the training data as well as the data supplied to the trained model to obtain the prediction, resulting in a simpler code when invoking the model.

Our pipeline will include the following stages:

1. Indexers
2. Encoder
3. Assembler
4. Label converter
5. Random Forest ML model

*More information on this and the Spark ML library can be found here: <https://spark.apache.org/docs/2.1.0/ml-features.html>*

#### 4.7.1.1 Indexers

First set up the indexers whose job is to encode a string column of labels to a column of label indices. We'll use it to encode our categorical column into a numerical value.

We use the StringIndexer which is a feature transformer. We use it to accomplish two things:

- Transforms the 'approved' column, which is a column of type 'string' containing only 'true' or 'false' values, into a numeric column, 'label', with '0' and '1' values so that the classifier can understand it.
- Transform the other categorical columns into a new set of index columns containing label indices. The indices are in the range of 0 to the number of labels for that category.

In [ ]:

```
# Instantiate the indexer for the approved column
approvalIndexer = StringIndexer(inputCol='approved', outputCol='label').fit(businessdata)
```



```
# Instanciate the rest of the indexers for the vehicle make, model and type columns.
indexers = [StringIndexer(inputCol=column, outputCol=column+"_index").fit(businessdata) for column in list(set(["vehicleMake", "vehicleModel", "vehicleType"])) ]

# Combine all indexers in a single list.
indexers.append(approvalIndexer)
```

#### 4.7.1.2 Encoder

We use the OneHotEncoder which is another feature transformer. It maps a column of label indices to a column of binary vectors, with at most a single one-value. Here we use it to map the index columns we created in the previous stage for a vehicle's make, model and type.

This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features such as *vehicleMake*, *vehicleModel* and *vehicleType*.

In [ ]:

```
encoders = [OneHotEncoder(inputCol=column+"_index", outputCol=column+"_encoded") for column in list(set(["vehicleMake", "vehicleModel", "vehicleType"])) ]
```

#### 4.7.1.3 Assembler

Next we set up a VectorAssembler which is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees.

We will train a random forest ML model so we will combine the features that we want to use in predicting our approval.

We first define the set of features that we will use to predict the the approved field. We use all fields except for approvedAmount since the Car Loan Approval process will not have that data when requesting a recommendation. However, based on the analysis in the previous section, it makes sense to include all other fields.

In [ ]:

```
# For categorical columns we use the column name produced by the onehotencoder: <colName>_encoded
features = ["creditScore", "requestedAmount", "vehicleMake_encoded", "vehicleModel_encoded", "vehicleType_encoded", "vehicleYear"]

assembler = VectorAssembler(inputCols=features, outputCol="features")
```

#### 4.7.1.4 Label converter

Finally, set up the IndexToString transformer which does the opposite of the StringIndexer. It maps a column of label indices back to a column containing the original labels as strings.

We use it to get the original label for our predicted value. In other words, map '1' or '0' to 'true' or 'false'.

In [ ]:

```
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel", labels=approvalIndexer.labels)
```

### 4.7.2 Creating the model

The model is built from the RandomForestClassifier algorithm. We chose to use Random Forest because it is flexible and easy to use. It produces great results even without hyperparameter tuning. These are parameters that get set before the training begins and are used to optimize the model produced.

In [ ]:

```
rf = RandomForestClassifier(labelCol="label", featuresCol="features")
```

In the cell below we split the data into training data and test data. The prediction model is then trained and tested, and finally the accuracy of the model is displayed.

In [ ]:

```
# Select the data
```

```
businessdata = businessdata[["creditScore", "requestedAmount", "vehicleMake", "vehicleModel", "vehicleType", "vehicleYear", "approved"]]
```

```
# Split the data into a training and testing set (80/20)
```

```
splitted_data = businessdata.randomSplit([0.8, 0.2], 24)
```

```
train_data = splitted_data[0]
```

```
test_data = splitted_data[1]
```

```
# Instanciate the pipeline object, specifying all the stages we defined above
```

```
pipeline = Pipeline(stages=indexers + encoders + [assembler, rf, labelConverter])
```

```
# Train the model
```

```
model = pipeline.fit(train_data)
```

```
# Test the model
```

```
predictions = model.transform(test_data)
```

```
# Compute the accuracy of the predictions
```

```
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
```

```
accuracy = evaluator.evaluate(predictions)
```

```
# Print the result
```

```
print("Accuracy = %g" % accuracy)
```

```
print("Test Error = %g" % (1.0 - accuracy))
```

## 4.8 Step 4: Store the model in Watson ML

Watson machine learning is used here to store the resulting model. After the model is stored, Watson machine learning makes it possible to create an HTTP scoring endpoint, which is then used as the recommendation service.

The code below stores the created model and pipeline in IBM Watson Machine Learning.

Lets start by importing the Watson Machine Learning API package.

Documentation on this API can be found here: <https://wml-api-pyclient.mybluemix.net/>

In [ ]:

```
from watson_machine_learning_client import WatsonMachineLearningAPIClient
```

Instantiate a Watson machine learning client. Note that if you are using your own instance of IBM Watson Machine Learning service you need to specify the authentication information for it in the cell below (wml\_credentials).

In [ ]:

```
# Authenticate to Watson Machine Learning service on IBM Cloud.
```

```
wml_credentials={
    "apikey": "8aSfy7WbWV_7ioSe8kR4_tEclghS6Z07wkIf1thGQmgT",
    "iam_apikey_description": "Auto generated apikey during resource-key operation for Instance - crn:v1:bluemix:public:pm-20:us-south:a/96f925a37d236c8abd126579c5a53a7b:8f794cbe-79cd-4d90-acb2-f797eca3dbec::",
    "iam_apikey_name": "auto-generated-apikey-109020c5-c94e-457e-9246-9ee6f63f3a62",
    "iam_role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Writer",
    "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity::a/96f925a37d236c8abd126579c5a53a7b::serviceid:ServiceId-547f2b67-3197-4cae-8348-b98c9784b4e9",
    "instance_id": "8f794cbe-79cd-4d90-acb2-f797eca3dbec",
    "password": "6800c9a6-ae6-4bea-a59d-09511ae025c5",
    "url": "https://us-south.ml.cloud.ibm.com",
    "username": "109020c5-c94e-457e-9246-9ee6f63f3a62"
}
```

```
client = WatsonMachineLearningAPIClient(wml_credentials)
```

We can now save the model and the training data. Call the store\_model API to store trained model into Watson Machine Learning repository on Cloud.

In [ ]:

```
published_model_details = client.repository.store_model(model=model, meta_props={'name': 'Recommendation Prediction Model'}, training_data=train_data, pipeline=pipeline)
```

## 4.9 Step 5: Deploy the model

Now that the model is stored, we need to deploy it in a runtime environment, we start by retrieving the model uid:

In [ ]:

```
model_uid = client.repository.get_model_uid(published_model_details)
print(model_uid)
```

We use the deployments client API to create a new deployment for our model:

In [ ]:

```
deployment_details = client.deployments.create(asset_uid=model_uid, name='Recommendation Prediction Model')
```

Your model has been deployed and it is ready for use. Take note of the model uid and deployment uid above. You will need these values for Section 3 of the lab. These will be used by a Service Flow to make the REST call to the scoring API and obtain a recommendation for the approved field.

## 4.10 Step 6: Testing the deployed model

You can test the model using the deployments client API.

The deployment details specifies the URL that will allow us to score against the published model.

In [ ]:

```
recommendation_url = client.deployments.get_scoring_url(deployment_details)
```

```
print(recommendation_url)
```

Test using different input value. For categorical variables you must pass a value that is known to the model. To quickly look at the different labels for each category you can use the following code to examine the Spark DataFrame.

In [ ]:

```
businessdata.groupBy("vehicleMake").count().show()
```

We now call the deployments.score API to get a prediction.

In [ ]:

```
import json
```

```
# Declare summary input for the prediction
```

```
recommendation_data = {"fields": ["creditScore", "requestedAmount", "vehicleMake", "vehicleModel", "vehicleYear", "vehicleType"],
                        "values": [[350, 14654, "GM", "Malibu", 2011, "Car"]]}
```

```

# Call the scoring API to predict the approval
scoring_response = client.deployments.score(recommendation_url, recommendation_data)
i = scoring_response['fields'].index('predictedLabel')
j = scoring_response['fields'].index('probability')
print("Recommend to approve = %s" %scoring_response['values'][0][i])
print("Confidence = %g" %scoring_response['values'][0][j][1])

# Uncomment to dump the full response
# print(json.dumps(scoring_response, indent=3))

```

## 4.11 Conclusion

Using this notebook you have trained and deployed a machine learning model that can provide a recommendation to approve or reject a car loan request. With the two ID's created in step 5, you can now continue to Section 3 in the lab instructions to complete the process application changes.