

Integrated Chatbot User Experience for the JD Edwards Administration

WHITE PAPER
SEPTEMBER 13, 2018

DISCLAIMER

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Table of Contents

OVERVIEW	6
Project	6
Purpose	6
Use Case	6
The JDE Solution	6
Demo Script/Interaction	6
Benefits	7
Minimum Technical Requirements	7
Create Your Own Intelligent Bot	8
What Are Intelligent Bots?	8
Overview of Bot Development	8
Create a BOT	8
Create BOT	9
Create Intents	9
Create Entities	10
Add Entities	11
Train the Bot	11
Which Training Model to Use?	11

Dialog Flow.....	12
Testing.....	14
Custom Component.....	15
How Do Custom Components Work?	15
The Component Service.....	15
The Shell	15
The Registry	15
Component Modules	16
The SDK.....	16
The Message Model.....	16
Accessing the Intelligent Bots SDK	16
Creating the Component Service in AMCe.....	17
Associating APIs with a Backend	24
Adding Custom Component in BOT Builder	26
Adding Custom Component to the Dialog Flow.....	27
Settings (Configure Channels).....	30
Embedding a Chat Bot in a JD Edwards E1Page	32
Prerequisites.....	32
Steps	32
Uploading Chat Bot as an E1 Page	40

OVERVIEW

Project

This case study highlights a possible JD Edwards feature regarding chatbots that would be built around the JDE administration. JD Edwards Development JDE-Labs is currently researching the use of chatbots within EnterpriseOne. This functionality is not generally available.

Purpose

To introduce basic concepts of chatbots and give hands-on experience of how to build an intelligent bot that is embedded within JD Edwards, enabling stakeholders to utilize the same to build their own bots for JD Edwards.

Use Case

There are certain actions JDE admins perform often. Sometimes these actions involve updates to multiple applications to complete the workflow. Creating users is one such use case which involves the following steps:

1. Add user record in Address Book.
2. Create a user profile.
3. Set up a role.
4. Assign environments to role.
5. Assign role to user.
6. Set up security for user.

Using the Oracle Intelligent BOT service and orchestrations, this action can be performed quickly using chat conversations. A similar model, the integrated modern chatbot interface technology, can be applied to several business process within JD Edwards. There are several repetitive steps, prone to manual user errors. In these cases, the chatbot interface could open a multi-step process using a progressive conversational interaction. Here, the user focuses only on inputs and the system executes the repetitive steps on the backend.

The JDE Solution

At a high level, the solution includes:

1. Chatbot interface for user interaction and experience, using the Intelligent Bots feature of the Autonomous Mobile Cloud Enterprise (AMCe).
2. JD Edwards Orchestrations to automate repetitive tasks.

Demo Script/Interaction

This is an example of a conversation using a chatbot (Edward) to automate the user creation process:

Admin: Create user John

Edward: What is the address book number? (Format: AB# 1234)

Admin: AB# 1007

Edward: What is the alpha name? (Format: Financial Distribution Company)

Admin: 'John D'

{After few seconds}

Edward: User creation successful

Benefits

Quicker turnaround time for business process completion.

Improved productivity and reduced error rates. The system performs the repetitive tasks, allowing the user to focus on exceptions and complex business conditions.

Minimum Technical Requirements

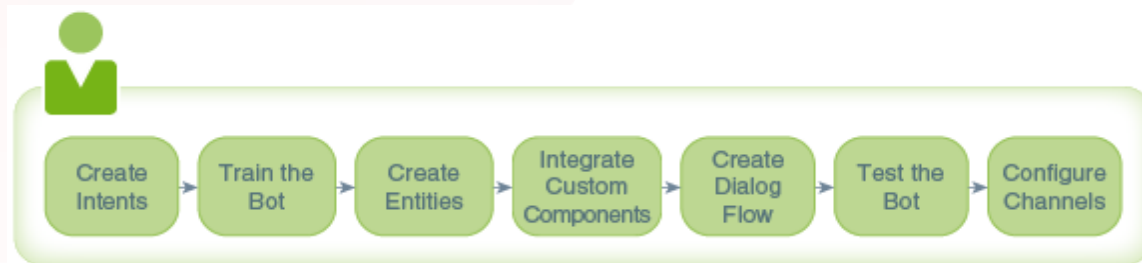
- Autonomous Mobile Cloud Enterprise (AMCe) instance – This is where we build the BOT and custom component.
- Node JS – To install all the required modules for the package.

CREATE YOUR OWN INTELLIGENT BOT

What Are Intelligent Bots?

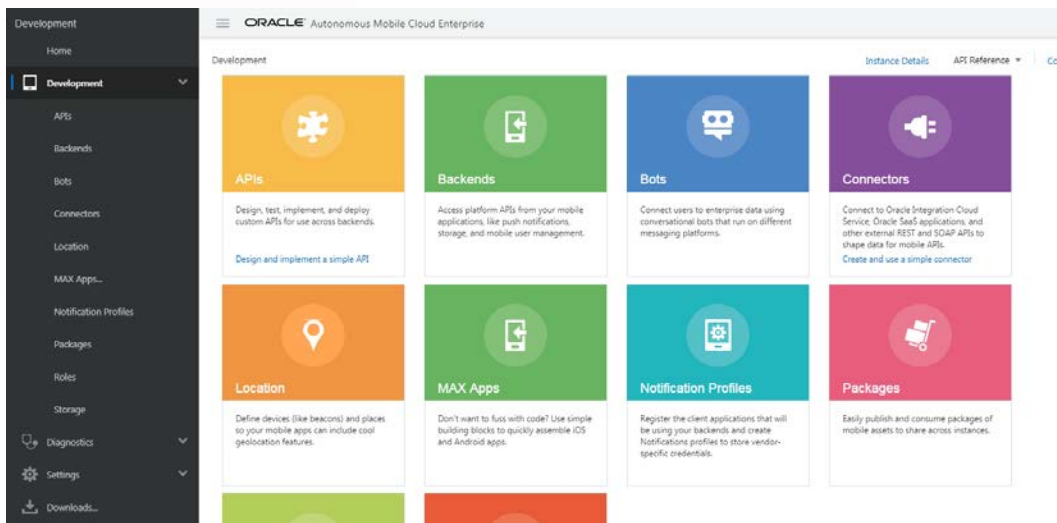
A bot is a virtual personal assistant that completes a task through a combination of text messages and simple UI elements like select lists. While a bot can open your enterprise to messaging, it is not a replacement for a mobile or web app. It instead provides a new channel.

Overview of Bot Development

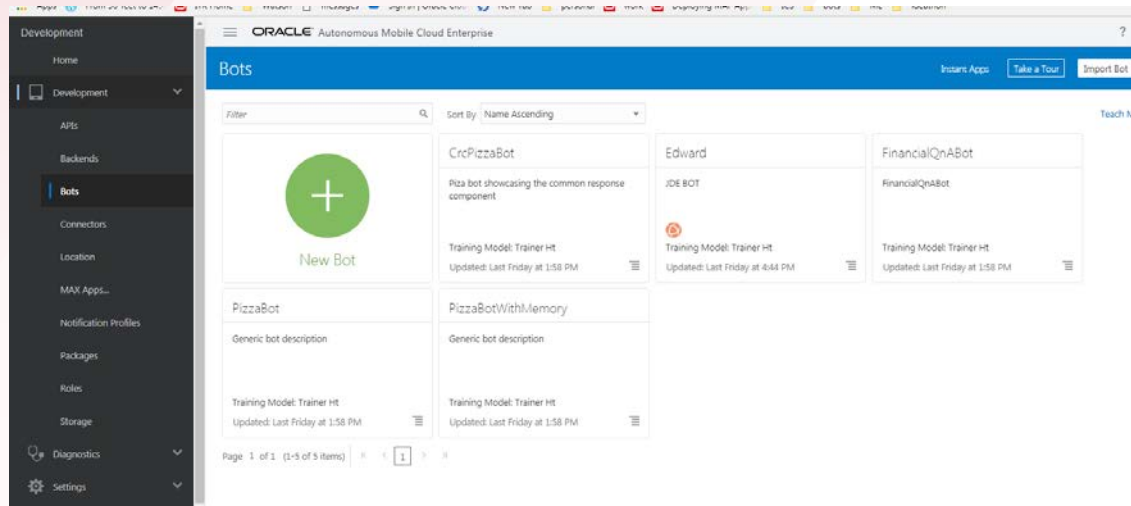


Create a BOT

1. Open your Autonomous Mobile Cloud Enterprise (AMCe) instance and click  to open the side menu.

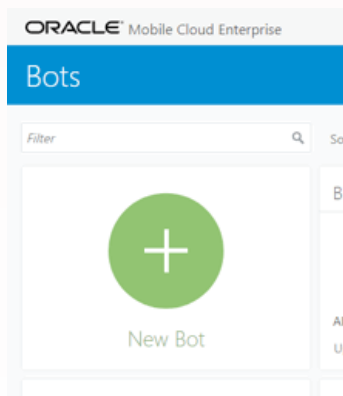


2. Select Development > Bots to open the Bot Builder, which you can use to create your own bot.



Create BOT

To create a new bot, click the New Bot button. Next, we need to introduce some basic information, such as the name of the chat and a brief description. In this particular case, we will create a very simple bot for a JDE admin. The bot will provide the JDE admin with the ability to create users with commands.

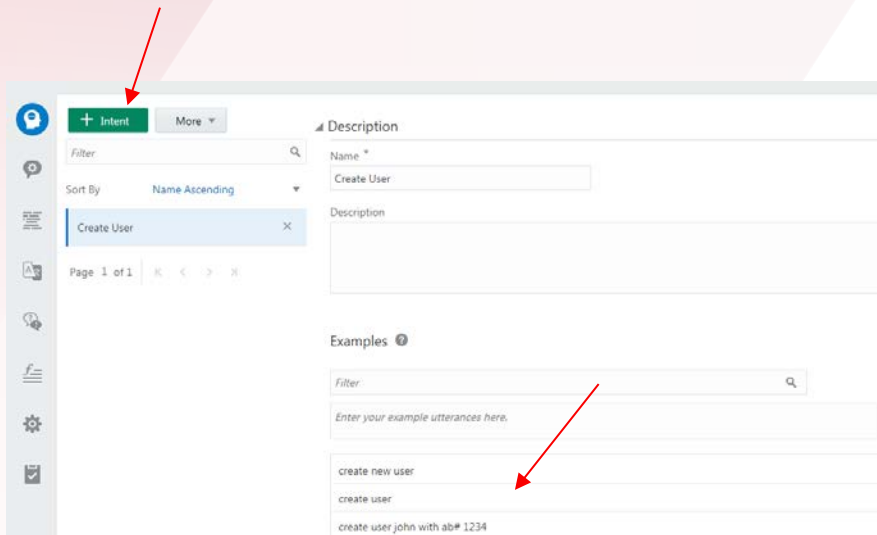


Create Intents

The first option is called Intents, which are categories of actions or tasks you expect your bot to perform. Here, we will record some questions/utterances about the topic so the chatbot can acquire the information it needs to fulfil the requirement. Click on the **Intent** button to create your first intent, and name it **Create User**.

Now we can create different utterances or statements to help the bot understand what the user needs. We shall add three utterances for our use case:

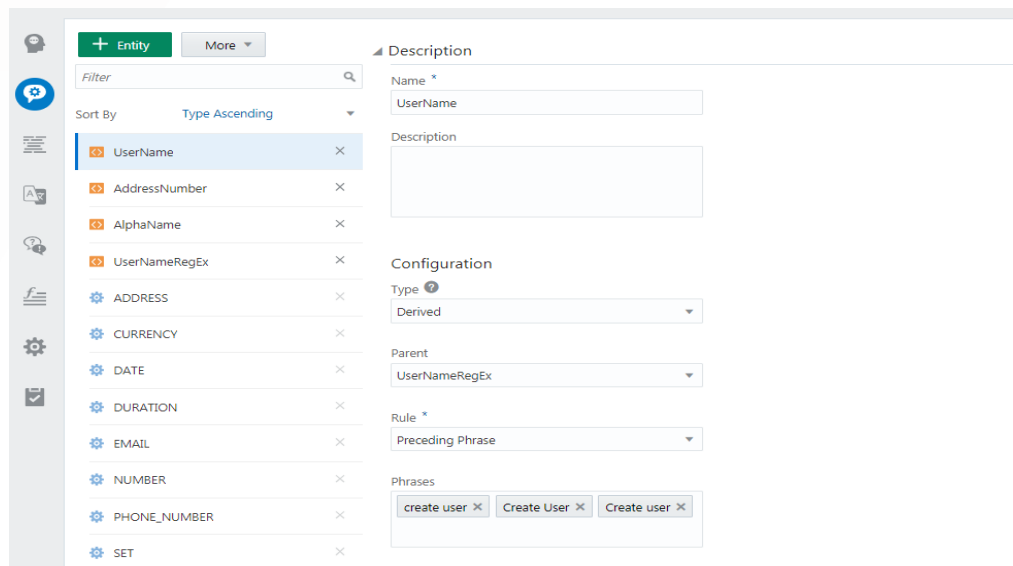
- » create user JDE with AB# 1003
- » create new user
- » create user



Create Entities

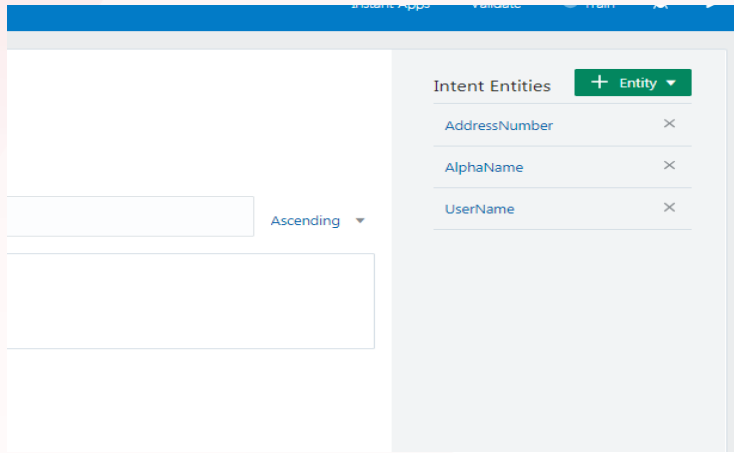
Entities are variables that identify key pieces of information from the user input that enable the bot to fulfill a task. Go to the Entities tab and create the following entities that are required to create a user:

ENTITY NAME	TYPE	VALUE
UserNameRegEx	Regular Expression	[a-zA-Z][a-zA-Z0-9]{2,7}
UserName	Derived	UserNameRegEx
AddressNumber	Regular Expression	[aA][Bb]# [0-9]{1,6}
AlphaName	Regular Expression	[a-zA-Z]{1,10}



Add Entities

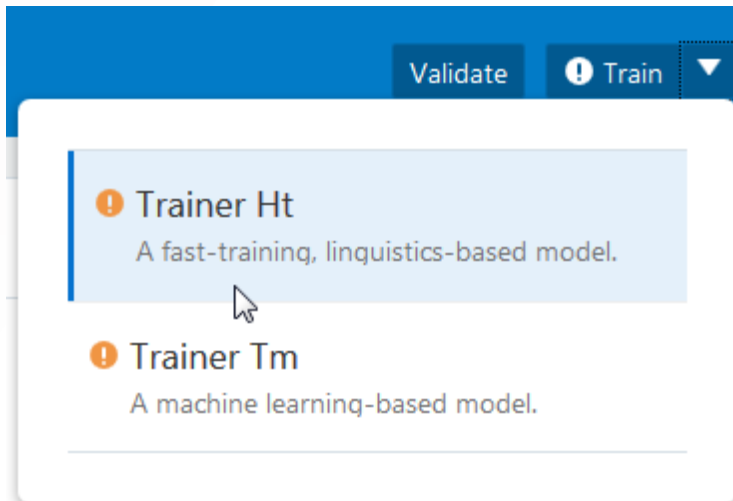
Add the above created entities to the Create User intent you previously created.



Train the Bot

To enable your bot to reference intents when it parses the user input, you need to train it. Specifically, you need to train it with the intents and their utterances (collectively, the training data), so that it can resolve the user input to one of the intents. Click Train, choose a model and then click Submit.

Note: You must have minimum of two intents to train the bot and each intent should have at least 3 utterances. You can create a dummy intent to meet this requirement if needed.



Which Training Model to Use?

- Create the initial training corpus.
- Train with Trainer Ht. You should start with Trainer Ht because it doesn't require a large set of utterances. As long as there are enough utterances to disambiguate the intents, your bot will be able to resolve user input.
- Refine your corpus, and retrain with Trainer Ht. Repeat as necessary. Training is an iterative process.
- Train with Trainer Tm. Use this trainer when you've accumulated a robust set of intents.

Dialog Flow

Next, you need to give the bot the ability to express its intelligence to its users by creating the dialog flow. The dialog flow describes how your bot reacts, as different intents are resolved. It defines what your bot says to its users, how it prompts them for input, and how it returns data. Think of the dialog flow as a flow chart that's been transposed to a simple markdown language. In Intelligent Bots, this markdown language is a version of YAML called BotML.

By default, the following template will be available:

```
1 #metadata: information about the flow
2 # platformVersion: the version of the bots platform that this flow was written to work with
3 metadata:
4   platformVersion: 1.0
5 main: true
6 name: Fresh_Bot
7 #context: Define the variables which will be used throughout the dialog flow here.
8 context:
9   variables:
10 #The syntax for defining the variables is variablename: "variableType".
11 # The "variableType" can be defined as a primitive type ("int", "string", "boolean"), "list", or an entity name. A variable can also hold the results returned by the Intent Engine. For these variables, the
12 # "variableType" must be "nlpresult" (for example, iResult: "nlpresult").
13   greeting: "string"
14   name: "string"
15 #states is where you can define the various states within your flow.
16 # The syntax for defining a state is
17 # stateName:
18 #   component: Specify the component you want to use. This can be either a Built-In or custom component.
19 #   properties:
20 #     property1: "value" (These are the properties to the specified component
21 #   transitions: You can specify one of the following four
22 #     next: Specify the state you want to execute next after this state. By default it will execute the state specified after this
23 #     error: Specify the state you want to execute in case the component encounters any error in execution.
24 #     actions: You can handle various actions returned by the components here the syntax is actionName: "stateName"
25 #     action1: state1
26 #     return: "done" You can exit the flow using the return statement
27 states:
28   askGreeting:
29     component: "System.List"
30     properties:
31       options: "Hello!, Ola!, Vannakam!, Namaste!"
32       prompt: "Hi there! What would you like me to echo back?"
33       variable: "greeting"
34   askName:
35     component: "System.Text"
36     properties:
37       prompt: "What is your name?"
38       variable: "name"
39   start:
40     component: "System.Output"
41     properties:
42       text: "${greeting.value} ${name.value}"
43     transitions:
44       return: "done"
```

Clear the existing content and paste the content inside the box below: **(Note: Indentation should be strictly followed)**

metadata:

platformVersion: "1.0"

main: true

name: "JDEBotMainFlow"

context:

variables:

iResult: "nlpresult"

```

states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.5
    transitions:
      actions:
        Create User: "createuserflow"
        unresolvedIntent: "unresolved"
  createuserflow:
    component: "System.Output"
    properties:
      text: "Sorry I am not yet programmed to create user"
    transitions:
      return: "createuserflow"
  unresolved:
    component: "System.Output"
    properties:
      text: "Sorry I don't understand that question!"
    transitions:
      return: "unresolved"

```

Now let's describe some of the variables included in the YAML.

The **iResult** variable will contain information and values generated by the natural language process engine. At the same time, it helps to determine the intents that trigger the execution of the Flow. This variable takes its value from **System.Intent**, which represents engine instance and the algorithm.

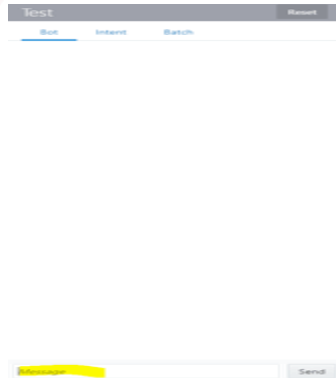
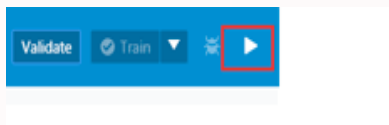
The **confidenceThreshold** variable is very important since it defines the precision value that the engine will use in order to determine what is going to be executed once the Intent has been evaluated. The engine and the algorithm will evaluate each user Intent and assign each a score; the score that is higher than the **confidenceThreshold** will be executed. Increasing or decreasing this value will impact the accuracy with which the bot can resolve the input from the user.

To validate the BotML, click on the **Validate** button in the upper right section of the Flow screen. You should see a confirmation message.

If there are two or more intents then we need to train the bot. To do that we need to click the Train button (upper right) so the bot can be trained with the Intents, Entities and Flow. Once we click Train we are ready to test the bot. In our case, as we have only one intent we can skip training the bot.

Testing

Click the Play button, located in the upper right part of the screen. Enter Create user. The system returns the message “Sorry I am not yet programmed to create user”. You get this message because the dialog flow is programmed to respond with this message for the create user intent.



CUSTOM COMPONENT

How Do Custom Components Work?

Your bot uses custom components when it needs to return data, execute business logic, or render channel-specific UI components like the carousel in Facebook Messenger.

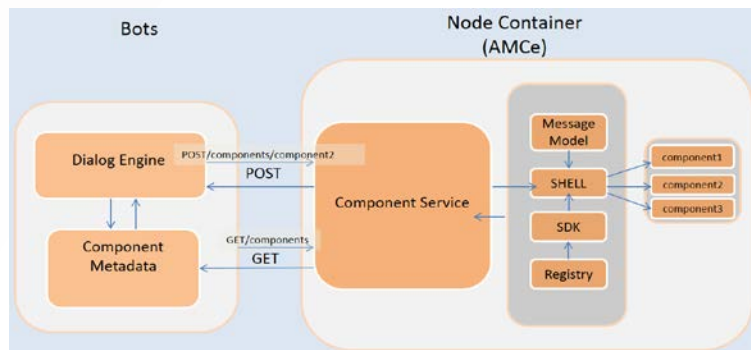
Like the built-in components, the custom components are re-usable units of work that you define within each state node of your dialog flow. Unlike the built-in components, custom components perform actions that are specific to your bot. They execute functions that the system components can't.

Custom components don't reside within Intelligent Bots. Their functionality is provided through backend services that are accessed through calls made to, and returned from, a REST service called the Component Service. As the Dialog Engine enters a state in the dialog flow, it assesses the component. When it encounters one of the built-in components (noted by System.), it executes one of the generic tasks described in Built-In Components: Properties, Transitions, and Usage. When the Dialog Engine discovers a custom component, however, it calls the Component Service, which hosts one or more custom components.

The Component Service first finds and then invokes the custom component on behalf of the Dialog Engine. When a custom component is invoked, it can pass input parameters to a backend service and return the result. The Dialog Engine then resumes, moving on to the next state in the dialog flow, or to the state dictated by the action described in the returned JSON payload.

The Component Service

The Component Service is hosted in its own Node container. As pictured here, the Node container can be part of AMCe, but it can be part of any other REST infrastructure as well.



The Shell

The Shell routes the GET and POST requests. It produces a list of components in response to the GET call made by Intelligent Bots when you register a Component Service. The Shell also invokes the component using the component name that's appended to the POST call (POST uri/components/{ComponentName}). To respond to these requests, the Shell component references a file in the Registry component that maps the component names to their corresponding JavaScript implementation files.

The Registry

The Registry component maps each component to its implementation. Within the Registry.js file, a JSON object definition surfaces the components to the Shell. Each component is described by a name-value pair in which the *name* is the name of the component (like 'Balance Retrieval' in the following import statement) and the *value* is a return function with a reference to the JavaScript module location relative to the Registry.js file (.). In this snippet, the two components, CreateUser and TrackSpending, are custom components, each of which map to a separate JavaScript module. The require function includes these separate modules in the Registry.js file.

```
'use strict';

module.exports = {

  'jde.CreateUser': require('./jde/create_user'),

  'TrackSpending': require('./banking/track_spending')

}
```

Component Modules

Each component is written as JavaScript module. The functions and business logic to be executed are written here.

The SDK

You can leverage the SDK, whose helper methods enable the components to access the context of a bot's request messages, which can be comprised of elements that describe the variable values, the language processing results, the extracted entities, and any input parameters that have been defined for the component. The SDK also enables the components to return a response to the bot.

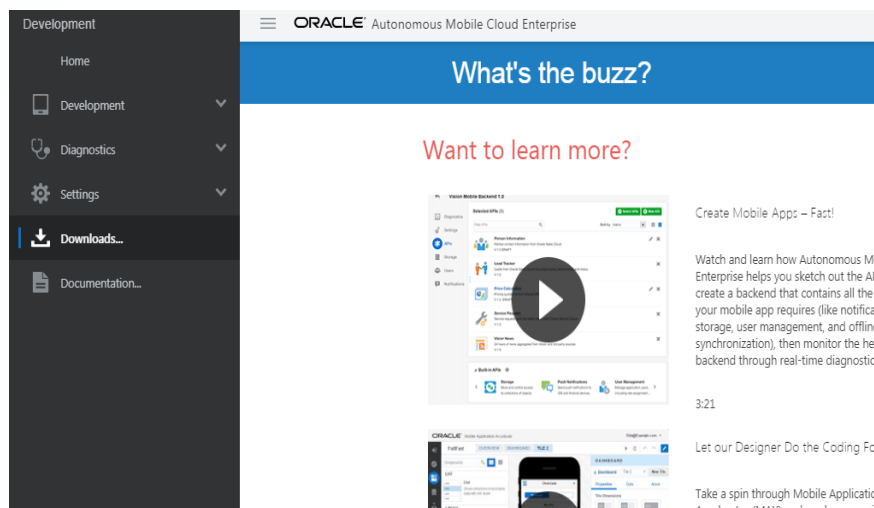
The Message Model

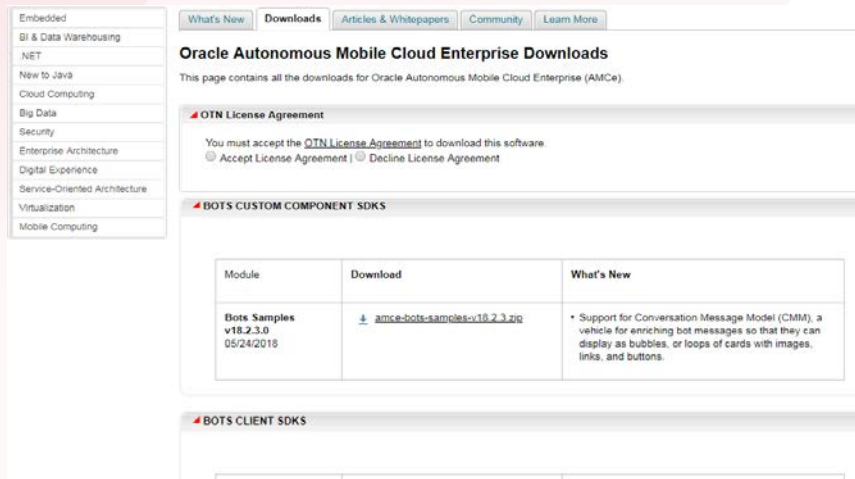
The Message Model is a utility class that creates and validates the message structure.

An instance of this class is instantiated with the payload that represents the message so that the message can be parsed and validated.

Accessing the Intelligent Bots SDK

To create your custom API you must first get the bot SDK. You can get the Intelligent Bots SDK (omce-bots-sdk-
<version_number>.zip) from the Oracle Technology Network's Oracle Autonomous Mobile Cloud Enterprise download page. You can also access this page by clicking Downloads in the left navigation bar.






After you unzip the file, open the API implementation folder. It contains the following artifacts that you modify to build your service. It includes JavaScript files for the Shell.

Registry and the SDK (shell.js, registry.js, and sdk.js). It also includes the following:

- mcebots.js: Contains the generic component logic. You copy and paste this into your own component service.
- package.json: Contains the node.js module dependencies required for the project's package.json file.
- mcsbots.raml: A template for creating the AMCe custom API.

Name	Date modified	Type
node_modules	5/31/2018 3:26 AM	File folder
pizza	5/24/2018 2:58 AM	File folder
mcebots.js	5/24/2018 1:38 AM	JScript Script File
mcebots.raml	5/24/2018 1:34 AM	RAML File
MessageModel.js	5/24/2018 2:58 AM	JScript Script File
package.json	5/24/2018 2:58 AM	JSON File
package-lock.json	5/24/2018 1:27 AM	JSON File
registry.js	5/24/2018 2:58 AM	JScript Script File
sdk.js	5/24/2018 2:58 AM	JScript Script File
shell.js	5/24/2018 2:58 AM	JScript Script File
swagger.json	5/24/2018 2:57 AM	JSON File
toolsConfig.json	5/24/2018 2:57 AM	JSON File

Creating the Component Service in AMCe

1. Define the GET and POST endpoints. You can define these endpoints on your own, or use the starter RAML template (mcebots.raml).
 - a. In AMCe, click  to open the side menu and select Development > APIs.
 - b. Click New API. Enter the API name (jde_bot), a description, and a short description.
 - c. Drag mcebots.raml into the dialog and then click Create. Make sure to rename the API to jde_bot.

New API

Either upload a valid RAML document to jumpstart your API creation, or enter the information below to get started.

* **API Display Name** 1.0


* **API Name**

https://8106505852EB4E01B309089D20233D48.mobile.ocp.oraclecloud.com:443/mobile/custom/jde_bot/

* **Short Description**

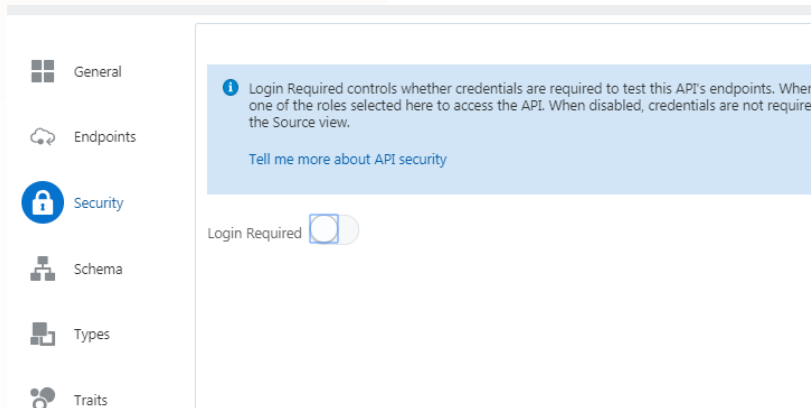
100 characters left

OR

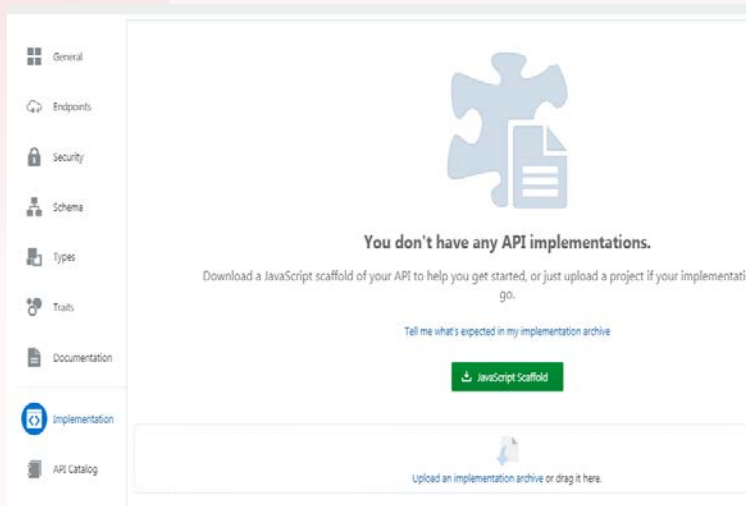
 Upload a RAML document or drag it here.

Create

2. To enable anonymous access, click Security in the left navigation bar and then switch off Login Required.



3. Click Save.
4. Download the JavaScript scaffold:
 - a. Click Implementation in the left navigation bar.



- b. Choose Download JavaScript Scaffold.
- c. Unzip the scaffold file. This file contains the following:
 - The component service file: This file, which is named after your API (i.e jde_bot.js), contains the REST endpoints defined for AMCe custom code APIs.
 - package.json: The project configuration file. It includes a list of module dependencies.

Name	Type	Compressed size
jde_bot.js	JScript Script File	
jde_bot.raml	RAML File	
package.json	JSON File	
ReadMe.md	MD File	
samples.txt	Text Document	
swagger.json	JSON File	
toolsConfig.json	JSON File	

5. Implement the Custom Component:

- a. Within the scaffold file, add the SDK, Registry, MessageModel and Shell modules.
- b. Implement the scaffold's JavaScript to add the custom component logic. To do this, you're going to replace most of the contents of the component service file (jde_bot.js) with those of the mcebots.js file from the Intelligent Bots SDK:
 - i. Open the component service file (jde_bot.js) in the JavaScript editor of your choice.
 - ii. Note the service.get function URI. It looks something like /mobile/custom/jde_bot/components.
 - iii. Delete all of the contents of the file except for the comments at the top of the file.

- iv. Open the mcebots.js file and then copy its contents to the component service file.
- v. Replace the value of const apiURL = '/mobile/custom/bots/components'; with the value of the service.get function. For example, const apiURL = '/mobile/custom/jde_bot/components';.
- vi. Make sure script Points to the right path of shell.js file.

```
var shell = require('./shell')();
```

- vii. Save the file.

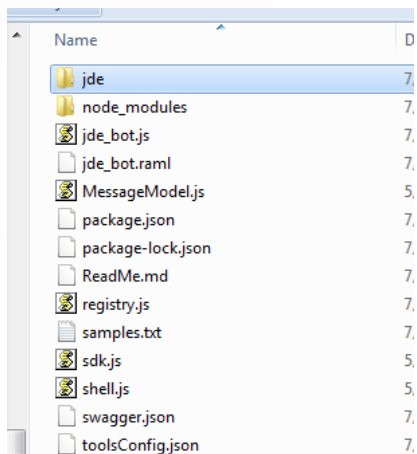
- c. Edit the package.json file in the scaffold file with the Bot SDK dependencies in the package.json file from the Intelligent Bots SDK:

- i. Open the Intelligent Bots SDK's package.json file in the text editor of your choice and then copy and paste its dependencies definition to a clipboard:

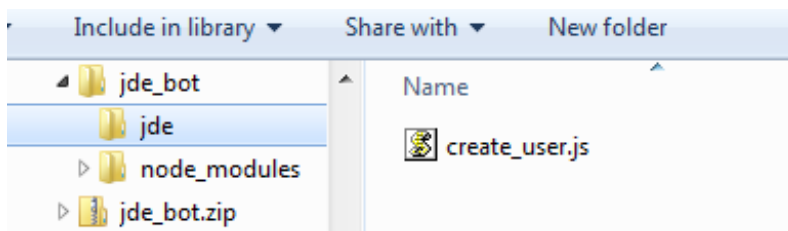
```
"dependencies": {
  "joi": "^9.2.0"
},
```

- ii. In the scaffold's package.json file, paste the definition on its own line, one directly after the "main":attribute.

- 6. Create a directory named jde within the scaffold file. Create the custom component module by creating a JavaScript file. This file includes the metadata and invoke functions.



- 7. Copy the below snippet to new javascript file within jde folder and name it create_user.js. Edit package.json file to add dependencies for the create_user.js file.



```

"use strict"

const axios = require('axios');

module.exports = {

  metadata: () => ({

    "name": "jde.CreateUser",      //component name

    "properties": {

      "userName": { "type": "string", "required": true },

      "addressNumber": { "type" : "string", "required": true },

      "alphaName": { "type" : "string", "required": true }

    },

    "supportedActions": [

    ]

  }),

  invoke: (conversation, done) => {

    var userName = conversation.properties().userName;

    var addressNumberStr = conversation.properties().addressNumber;

    // strip out prefix "AB# "

    var addressNumber = addressNumberStr.replace(/AB# /i, '');

    var alphaNameStr = conversation.properties().alphaName;

    // strip out single quotes

    var alphaName = alphaNameStr.replace(/'/g, '');

    conversation.logger().info('jde.CreateUser: creating JDE user ' + userName);

    var baseUrl = "http://machineip:port/jderest"; //add the url for your rest server

    var url = baseUrl + "/tokenrequest";

    console.log(url);

    //add user name and password to your rest server
  }
}

```

```

    axios.post(url, {username: 'username', password: 'pwd'})
    .then((response) => {
        var jsonData = JSON.stringify(response.data, null, '\t');
        var json = JSON.parse(jsonData);
        var token = json.userInfo.token;
        //ND_AddUser is the orchestration
        var url = baseURL + "/orchestrator/ND_AddUser";
        console.log(url);
        var params = {};
        var inputs = []
        params.token = token;
        params.inputs = inputs;
        params.inputs.push({"name": "AddressNumber", "value": addressNumber});
        params.inputs.push({"name": "AlphaName", "value": alphaName});
        params.inputs.push({"name": "SearchType", "value": "E"});
        params.inputs.push({"name": "BusinessUnit", "value": "1"});
        params.inputs.push({"name": "UserID", "value": userName});
        params.inputs.push({"name": "Role", "value": "SYSADMIN"});
        params.inputs.push({"name": "DataSource", "value": "DEFAULT"});
        params.inputs.push({"name": "SystemUser", "value": "JDE"});
        params.inputs.push({"name": "Password", "value": "welc2jde"});
        console.log(params);
        return axios.post(url, params); // using response.data
    })
    .then((response) => {
        console.log('Response', response);
        conversation.reply({ text: 'User ' + userName + ' created successfully'});
        conversation.transition();
        done();
    })

```

```

        .catch(function (error) {
            console.log(error);
            conversation.reply({ text: 'User ' + userName + ' creation failed' });
            conversation.transition();
            done();
        });
    }
};

```

The above file receives the entity values from the chat conversation and performs AIS calls to the orchestration ND_ADDUSER (Export file is available in lab_resources folder) with required inputs. The response received is then modified to user readable form and sent as reply to the conversation flow.

ND_ADDUSER orchestration contains service requests which performs following tasks:

SERVICE REQUESTS	INPUTS
Add user record	Addressnumber, AlphaName, SearchType (E), BusinessUnit (1)
Create user profile	UserId, AddressNumver
Assign role to user	UserID, Role (SYSADMIN)
Set up security for user	UserID, DataSource (Default), SystemUSer, Password

For more information on orchestrations, please refer to the [JD Edwards EnterpriseOne Tools Orchestrator Guide](#)

8. Edit package.json file to add the dependencies for create_user.js.

```

"dependencies": {
    "joi": "^9.2.0",
    "body-parser": "^1.15.0",
    "express": "^4.13.4",
    "http-auth": "^3.1.3",
    "moment": "^2.16.0",

```

```

    "request": "2.73.0",
    "twilio": "2.11.1",
    "jsdoc": "^3.4.3",
    "axios": "^0.12.0"
  },

```

9. Edit the registry.js file with the name and location of the component file.

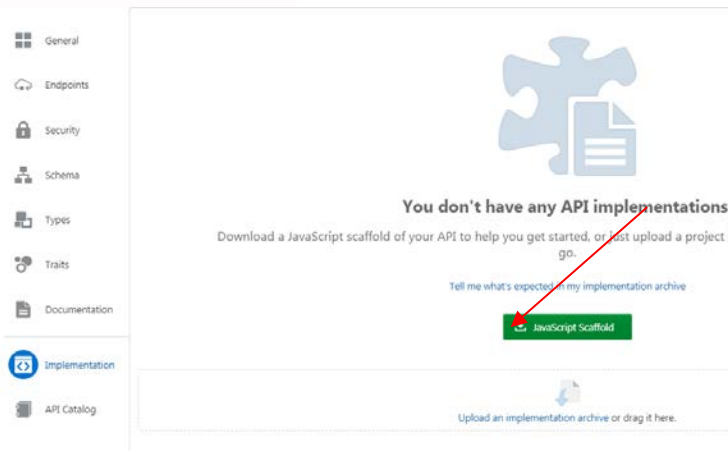
```

module.exports = {
  components: {
    // JDEBot
    'jde.CreateUser': require('./jde/create_user')
  }
};

```


10. Install the node module. Navigate to the scaffold file directory in command line and run NPM install.

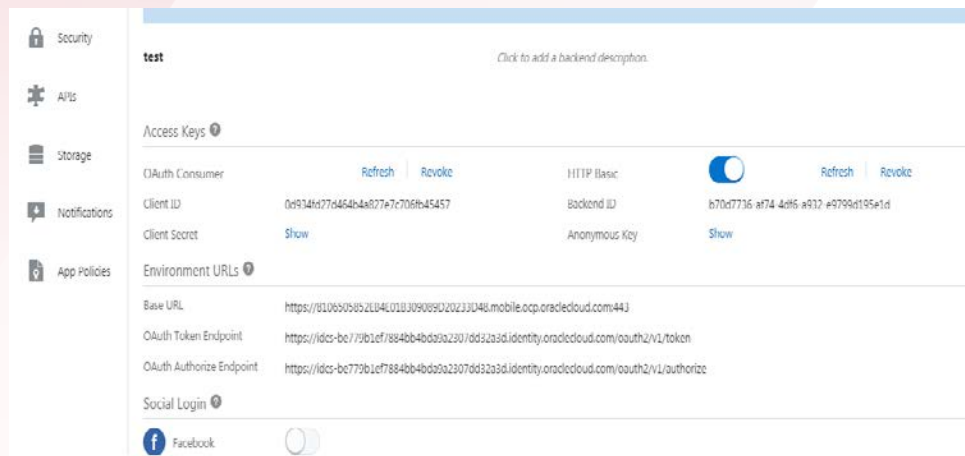
11. Package the scaffold file to .zip and upload the node project to AMCe.



Associating APIs with a Backend

You create a backend to serve as a secure gateway between your app and AMCe features, such as BOT and custom APIs. For your BOT to access these resources, it authenticates with a backend.

1. Click  to open the side menu and select Development > Backends.
2. Click New Backend.
3. Once you complete the dialog and the backend is created, keep the Settings page open.
4. The following authentication and connection details are generated when you create a backend and are displayed on the backend's **Settings** page:



Access Keys

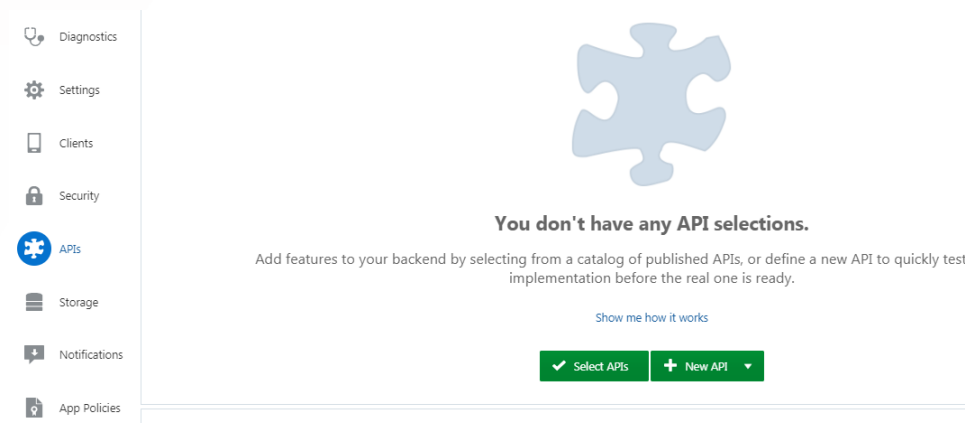
You can use these to control access to the backend. They are unique for each backend.

- OAuth Consumer keys are generated in the form of a client ID and a client secret.
- HTTP Basic Authentication keys are generated for you in the form of a backend ID and an anonymous key. (We will be using this one for our example.)

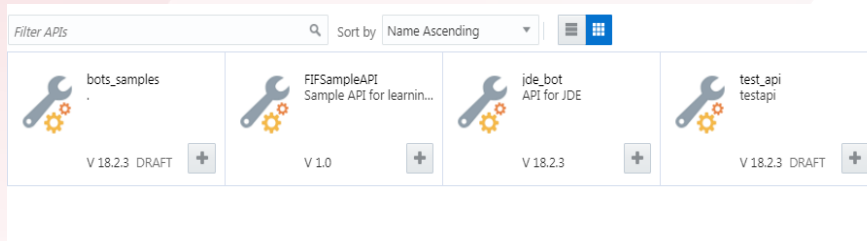
Environment URLs

- The Base URL is needed for all API calls. This URL is unique for each instance that you have provisioned.
- The OAuth Token Endpoint is the URL that your app needs to make OAuth token requests.
- The OAuth Authorize Endpoint is the URL that your app can use to get an authorization code to exchange for an OAuth access token.

5. In the left navigation bar, click **APIs**.



6. Click Select API and choose the API created in the previous section (i.e., jde_bot) using the + (Add) icon.



Adding Custom Component in BOT Builder

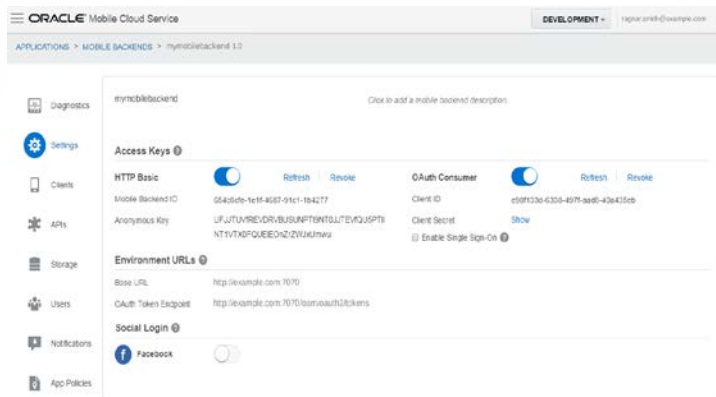
Navigate back to Development-> BOTS -> your bot (jde_bot) to register the component service with Intelligent Bots so that it can be discovered by the Dialog Engine.

1. In the left navigation bar, click Components ().
2. Click Add Service to open the Create Service dialog.

3. Add the name for the custom component service and an optional description.
4. Choose an authentication option:

Mobile Cloud: For authentication handled by a backend in AMCe.

This is the default setting. To complete the dialog, you need to reference the Settings page for the backend that hosts the API that implements the Custom Component Service. Authentication and Connection Info in *Developing Applications with Oracle Autonomous Mobile Cloud Enterprise* describes this page.

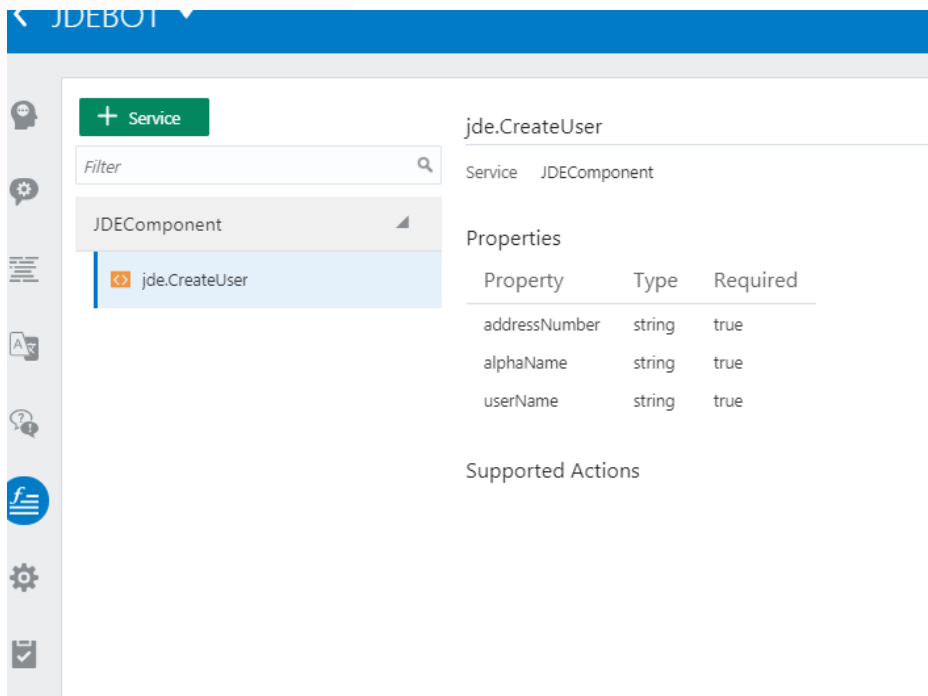


Metadata URL: This is custom API URL, which is displayed in the Overview panel of the API Designer. You should append /components to the API url.



User name and password: If you selected Use anonymous Access, you must provide the Anonymous Key. This value is generated when you create a backend, and is displayed on the Settings page for the backend that manages your API.

You are now ready to add the custom components to your OBotML definition. You can see your custom components loaded as shown below:



Adding Custom Component to the Dialog Flow

Navigate to Dialog Flow and replace the contents with the yaml code below. Also, replace the highlighted component name with your component name. Validate the yaml code with the Validate button at the top right. **(Note: Indentation should be strictly followed)**

```
metadata:
  platformVersion: "1.0"
main: true
name: "JDEdwardsBotMainFlow"
context:
  variables:
    iResult: "nlpresult"
    userName: "UserName"
    addressNumber: "AddressNumber"
    alphaName: "AlphaName"
    txnState: "string"
states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.5
    transitions:
      actions:
        Create User: "startCreateUserFlow"
        unresolvedIntent: "unresolved"
  startCreateUserFlow:
    component: "System.SetVariable"
    properties:
      variable: "userName"
      value: "${iResult.value.entityMatches['UserName']}[0]}"
    transitions: {}
  setAddressNumber:
    component: "System.SetVariable"
```

```

properties:
  variable: "addressNumber"
  value: "${iResult.value.entityMatches['AddressNumber']}[0]}"
  transitions: {}
setAlphaName:
  component: "System.SetVariable"
  properties:
    variable: "alphaName"
    value: "${iResult.value.entityMatches['AlphaName']}[0]}"
    transitions: {}
askUserName:
  component: "System.Text"
  properties:
    prompt: "What's the user name?"
    variable: "userName"
    maxPrompts: 1
  transitions:
    actions:
      cancel: "intent"
askAddressNumber:
  component: "System.Text"
  properties:
    prompt: "What's the Address Number (format AB# 1234)?"
    variable: "addressNumber"
    maxPrompts: 1
  transitions:
    actions:
      cancel: "intent"
askAlphaName:
  component: "System.Text"

```

```

properties:
  prompt: "What's the AlphaName (format 'Financial Distribution Company')?"
  variable: "alphaName"
  maxPrompts: 1
transitions:
  actions:
    cancel: "intent"
execCreateUser:
  component: "jde.CreateUser"
  properties:
    userName: "${userName.value}"
    addressNumber: "${addressNumber.value}"
    alphaName: "${alphaName.value}"
  transitions:
    return: "execCreateUser"
unresolved:
  component: "System.Output"
  properties:
    text: "Sorry I don't understand that question!"
  transitions:
    return: "unresolved"

```

Settings (Configure Channels)

Bots aren't apps that you download from an app marketplace. Instead, users access them through messaging platforms or through client messaging apps. Channels, which are platform-specific configurations, allow this access. A single bot can have several channels configured so that it can run on different services simultaneously.

Go to the Settings tab and add a new web channel as shown below. This will generate an app Id that can be used to call the BOT service from web.

Create Channel

* Name

Channel name

Description

Optional short description for this channel

Channel Type

Facebook Messenger

* Page Access Token

Facebook Messenger

Webhook

Web

iOS

Android

* App Secret

Copy from the Facebook app to here

Session Expiration (minutes)

60

Default

Channel Enabled

Create

General Channels Agent Integrations Q&A Routing Config

+ Channel

System_Bot_Test

Web

* Name

Web

Description

Optional short description for this channel

Channel Type

Web

App Id

5b190522f3454600219c2054

App Token

1/8sddczp8vonnqiahyIOa3d

Session Expiration (minutes)

60

Default

Channel Enabled

EMBEDDING A CHAT BOT IN A JD EDWARDS E1PAGE

Prerequisites

1. To get the OMCE client SDK for JavaScript, go to the Oracle Technology Network's Oracle Mobile Cloud Enterprise Downloads page and download the following archives:
 - a. OMCE Bots Client SDK for JavaScript v18.2.3.0
 - b. OMCE Bots Client samples for JavaScript v18.2.3.0
2. Install Node.js on your machine <https://nodejs.org/en/download/>.

Steps

Configure the SDK

1. Copy bots-client-sdk-js-18.2.3.0.zip to a Linux machine and extract the zip.
2. Run the configuration script. This script moves the required sdk file to one folder.

```
[opc@jdelabs bots_sdk_external]$ ./configure bots-client-sdk-js
```

```
Done! Files are available in /mnt/store/bots_sdk_external/bots-client-sdk-js_
```

3. Zip up the created directory and copy it back to your Windows machine.

```
mv bots-client-sdk-js_ bots-client-sdk-js
```

```
tar -cvzf bots-client-sdk-js.tar.gz bots-client-sdk-js
```

Configure the Sample

1. Extract bots-client-sdk-js-samples-18.2.3.0.zip on your Windows machine.
2. Delete the "bots-client-sdk-js" directory from "bots-client-sdk-js-samples-18.2.3.0\chat-sample-web\app".
3. Extract bots-client-sdk-js.tar.gz to the app directory ("bots-client-sdk-js-samples-18.2.3.0\chat-sample-web\app\bots-client-sdk-js").
4. Delete index.html from the app directory.
5. Replace the contents of home.html with the following snippet: Customizing the UI to rename the bot and removing the unwanted part of UI available in sample.

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
<meta charset="UTF-8">
```

```
<title>BotsChat Web</title>
```

```
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
<link rel="icon" type="image/x-icon" href="favicon.ico">
```

```
<link rel="stylesheet" href="styles/bootstrap.min.css">
```



```

    <link href="styles/style.css" rel="stylesheet">
</head>

<body>

    <div id="loader">

    </div>

    <div class="row app-header">

        <div class="col-xs-2"><a href="settings.html"></a></div>

        <div class="col-xs-6 col-xs-offset-1 app-title">JD Edwards Virtual Assistant</div>

    </div>

    <div class="container" id="configuration-form">

        <form class="form-signin text-center">

            <div class="row button-row">

                <button id="clearchatbtn" style="display: none;" class="btn btn-lg btn-primary
col-xs-8 col-xs-offset-2 col-md-4 col-md-offset-4" onclick="clearChat(event)">Clear
Chat</button>

            </div>

        </form>

    </div>

    <div class="row text-center">

        <div class="col-md-4 col-md-offset-4 footer-logo">

        </div>

    </div>

    <script src="scripts/app.js"></script>

</body>

</html>

```

The differences between the modified and original home.html file are shown below:

Original home.html	Modified home.html
<pre> <title>BotsChat Web</title> <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no"> <link rel="icon" type="image/x-icon" href="/favicon.ico"> <link rel="stylesheet" href="styles/bootstrap.min.css"> <link href="styles/style.css" rel="stylesheet"> </head> <body> <div id="loader"> </div> <div class="row app-header"> <div class="col-xs-12"></div> </div> <div class="col-xs-6 col-xs-offset-1 app-title">JD Edwards Virtual Assistant</div> </div> <div class="container" id="configuration-form"> <form class="form-signin text-center"> <div class="row"> <div class="col-md-4 col-md-offset-4 mcs-logo"> </div> </div> <div class="row"> <div class="col-md-6 col-md-offset-3"> <h2 class="title">Mobile Cloud Enterprise </h2> <h3 class="form-signin-heading">CLOUD</h3> <div id="indexpage"> <h2 class="welcome">Welcome</h2> <p>This sample app shows how to use the Oracle Intelligent BOT SDK to embed chat function in a web app. <p style="margin: 20px;">Your mobile app would go here!</p> </div> </div> </div> <div class="row submit-button"> <button id="openChatButton" class="btn btn-lg btn-primary col-xs-8 col-xs-offset-2 col-md-4 col-md-offset-4"> </div> <div class="row button-row"> <button class="btn btn-lg btn-primary col-xs-8 col-xs-offset-2 col-md-4 col-md-offset-4" onclick="clearChat"> </div> </form> </div> <div class="row text-center"> <div class="col-md-4 col-md-offset-4 footer-logo"> </div> </div> <script src="scripts/app.js"></script> </body> </pre>	<pre> <title>BotsChat Web</title> <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no"> <link rel="icon" type="image/x-icon" href="/favicon.ico"> <link rel="stylesheet" href="styles/bootstrap.min.css"> <link href="styles/style.css" rel="stylesheet"> </head> <body> <div id="loader"> </div> <div class="row app-header"> <div class="col-xs-12"></div> </div> <div class="col-xs-6 col-xs-offset-1 app-title">Chat SDK Sample</div> </div> <div class="container" id="configuration-form"> <form class="form-signin text-center"> <div class="row"> <div class="col-md-4 col-md-offset-4 mcs-logo"> </div> </div> <div class="row"> <div class="col-md-6 col-md-offset-3"> <h2 class="title">Mobile Cloud Enterprise </h2> <h3 class="form-signin-heading">CLOUD</h3> <div id="indexpage"> <h2 class="welcome">Welcome</h2> <p>This sample app shows how to use the Oracle Intelligent BOT SDK to embed chat function in a web app. <p style="margin: 20px;">Your mobile app would go here!</p> </div> </div> </div> <div class="row submit-button"> <button id="openChatButton" class="btn btn-lg btn-primary col-xs-8 col-xs-offset-2 col-md-4 col-md-offset-4"> </div> <div class="row button-row"> <button class="btn btn-lg btn-primary col-xs-8 col-xs-offset-2 col-md-4 col-md-offset-4" onclick="clearChat"> </div> </form> </div> <div class="row text-center"> <div class="col-md-4 col-md-offset-4 footer-logo"> </div> </div> <script src="scripts/app.js"></script> </body> </pre>

- Save the below image to directory "bots-client-sdk-js-samples-18.2.3.0\Chat-sample-web\app\images" as edward.png



7. Replace the contents of app.js located in the scripts directory with the following snippet:

```
!function(e,t,n,r){

    function s(){try{var e;if(("string"===typeof
this.response?JSON.parse(this.response):this.response).url){var
n=t.getElementsByTagName("script")[0],r=t.createElement("script");r.async=!0,r.src=e.url,n.paren
tNode.insertBefore(r,n)}}catch(e){}}var o,p,a,i=[],c=[];e[n]={init:function(){o=arguments;var
e={then:function(t){return c.push({type:"t",next:t}),e},catch:function(t){return
c.push({type:"c",next:t}),e}};return
e},on:function(){i.push(arguments)},render:function(){p=arguments},destroy:function(){a=argument
s}},e.__onWebMessengerHostReady__=function(t){if(delete
e.__onWebMessengerHostReady__,e[n]=t,o)for(var r=t.init.apply(t,o),s=0;s<c.length;s++){var
u=c[s];r="t"===u.type?r.then(u.next):r.catch(u.next)}p&&t.render.apply(t,p),a&&t.destroy.apply(t
,a);for(s=0;s<i.length;s++)t.on.apply(t,i[s])};var u=new
XMLHttpRequest;u.addEventListener("load",s),u.open("GET",r+"/loader.json",!0),u.responseType="js
on",u.send()}

(window,document,"Bots","bots-client-sdk-js");

Bots.on('message:received', function(message) {

    console.log('the user received a message', message);

});

Bots.on('message:sent', function(message) {

    if(message.text.match(/clear chat/i) != null){

        document.getElementById("clearchatbtn").click();

    }

});

function loadAppId(){

    var appId = window.localStorage.getItem("appId");

    if(appId){

        document.getElementById("appId").value = appId;

    }

}

function saveAppId(e){
```

```

e.preventDefault();

let appId = document.getElementById("appId").value;
console.log('Validate appId', appId);

// validate app id
initBots(appId)

    .then(function () {

        console.log('AppId is valid');

        window.localStorage.setItem("appId", appId);

        window.location.href = "home.html";

        document.getElementById("loader").style.display = "none";

    })

    .catch(function (err) {

        document.getElementById("loader").style.display = "none";

        document.getElementsByClassName("error")[0].style.display = 'block';

        console.log('AppId validating error', err);

    });

}

function loadChat(e){

    e.preventDefault();

    console.log('Init Bots SDK');

    var appId = window.localStorage.getItem("appId");

    initBots(appId)

        .then(function () {

            console.log("init complete");

            document.getElementById("loader").style.display = "none";

            Bots.open();

            document.getElementById("openChatButton").setAttribute("disabled", true)

        })

```

```

        .catch(function (err) {
            console.log(err);
        });
    }

function clearChat(e){
    e.preventDefault();

    var keys = Object.keys(localStorage);
    for(var i = 0; i < keys.length; i++){
        if(keys[i] === 'appId'){
            continue;
        }

        localStorage.removeItem(keys[i]);
    }

    location.reload();
}

function initBots(appId){
    return Bots.init({
        appId: appId,
        businessName: 'Edward', //name of your bot
        businessIconUrl: 'images/edward.png', //path to your image icon for your bot

        customColors: {
            brandColor: '286090',
        },
        customText: {
            headerText: '',
            introductionText: 'What can I help you with?',
        }
    });
}

```

```

    }).then(function (res){
        Bots.updateUser(
            {
                "givenName": "John",
                "surname": "Snow",
                "email": "john.snow@winterfell.com",
                "properties": {
                    "smoochCustomVariable1": "Lord",
                    "smoochCustomVariable2": "Commander"
                }
            }
        ).catch(function (err) {
            console.error(err);
        });
    });
}

let appId = "<your-app-ID>";

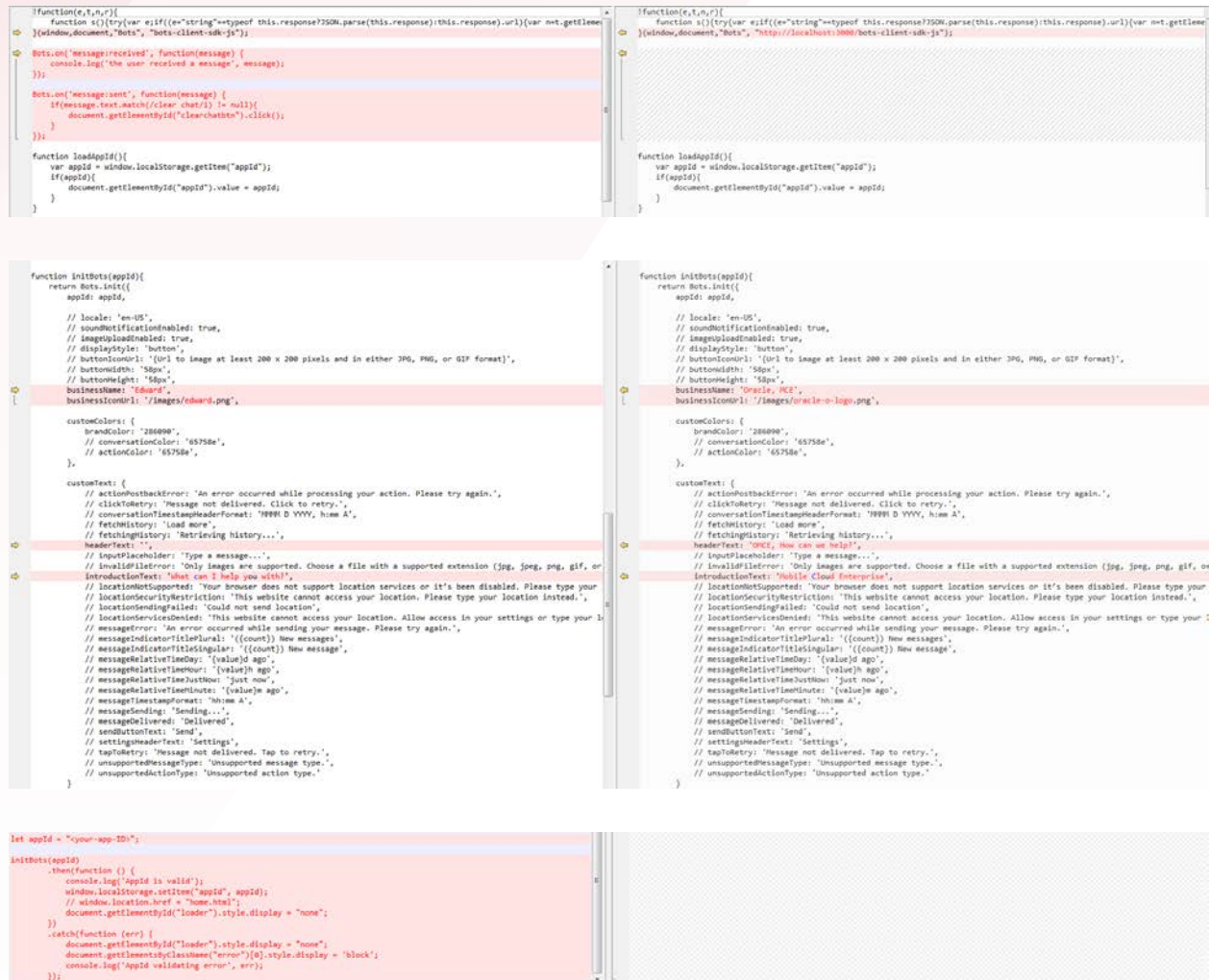
initBots(appId)

    .then(function () {
        console.log('AppId is valid');
        window.localStorage.setItem("appId", appId);
        // window.location.href = "home.html";
        document.getElementById("loader").style.display = "none";
    })

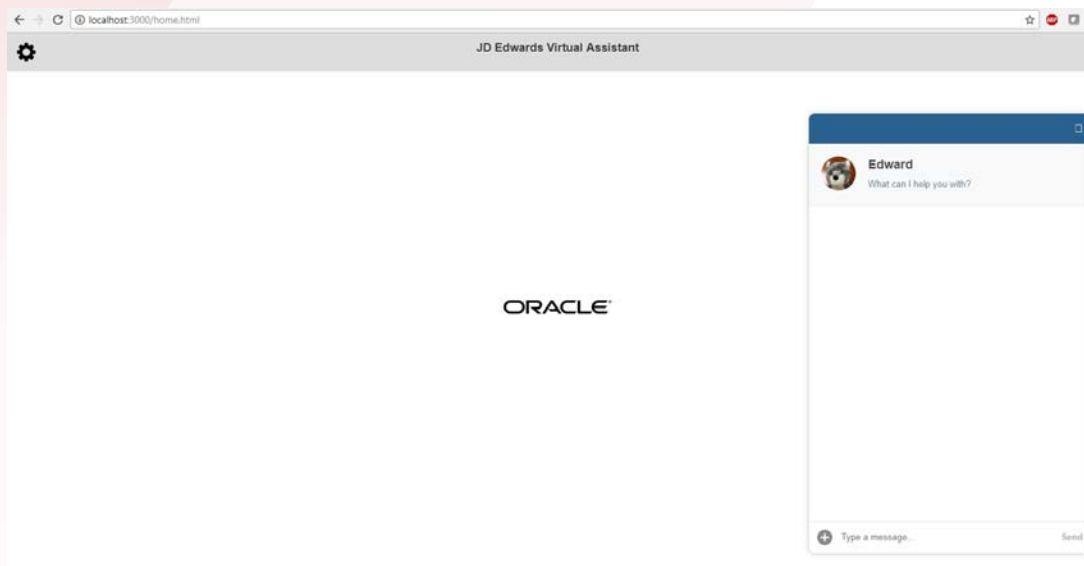
    .catch(function (err) {
        document.getElementById("loader").style.display = "none";
        document.getElementsByClassName("error")[0].style.display = 'block';
        console.log('AppId validating error', err);
    });
}

```

The comparison between the modified and original app.js file is shown below:

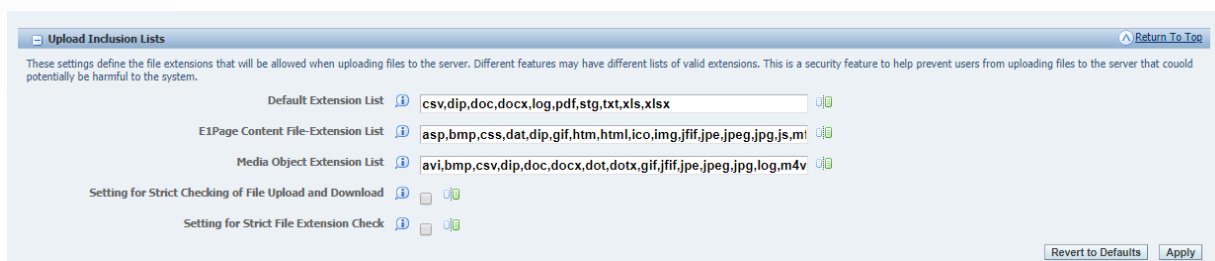


8. In app.js, replace "<your-app-ID>" with the Web channel App ID.
9. Navigate to chat-sample-web directory.
 - Run 'npm install'
 - Run 'node server.js'
 - In your browser, open <http://localhost:3000/home.html>
10. The chat interface will be initialized and connected to your application. Click the chat button at the bottom right of the window to open the chat interface.



Uploading Chat Bot as an E1 Page

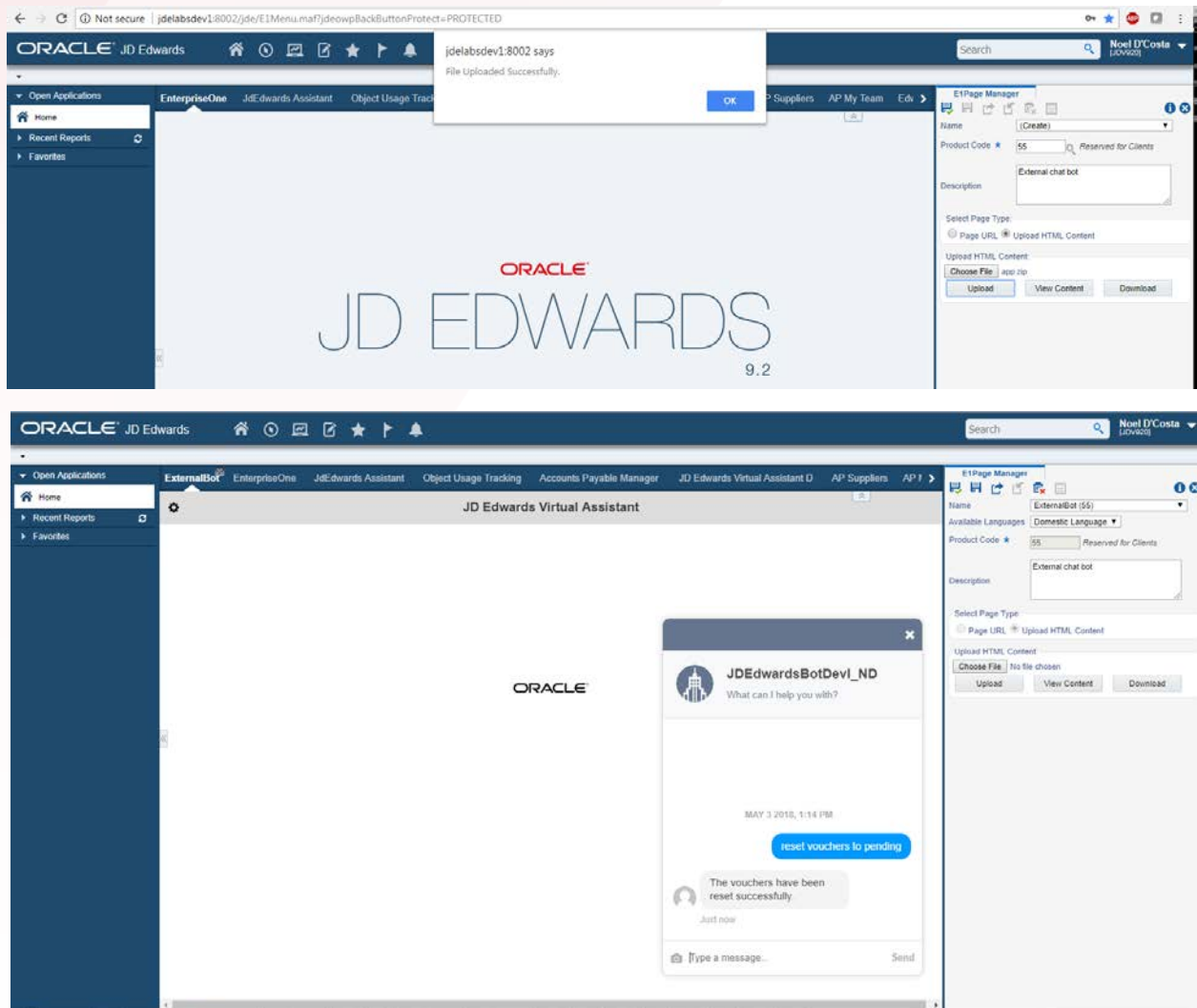
1. In Server Manager Console, edit the JAS ini settings under Security as shown below:
 - a. E1Page Content File-Extension List:
asp,bmp,css,dat,dip,gif,htm,html,ico,img,jfif,jpe,jpeg,jpg,js,mf,pdf,png,svg,tif,tiff,xml,zip,mp3,json,woff,eot,md,ttf,woff2
 - b. Setting for Strict Checking of File Upload and Download: Disabled
 - c. Setting for Strict File Extension Check: Disabled
2. Restart JAS after making the changes.



3. Navigate to directory "bots-client-sdk-js-samples-18.2.3.0\Chat-sample-web\app" and zip up the contents of the folder as an archive, say, app.zip.

Name	Date modified	Type	Size
bots-client-sdk-js	5/3/2018 12:30 PM	File folder	
images	5/3/2018 12:47 PM	File folder	
scripts	5/3/2018 12:41 PM	File folder	
styles	1/19/2018 4:02 PM	File folder	
app.zip	5/3/2018 1:10 PM	zip Archive	703 KB
home.html	5/3/2018 12:36 PM	Chrome HTML Do...	2 KB
settings.html	1/19/2018 4:02 PM	Chrome HTML Do...	2 KB

4. Login to JAS and upload the zip as an E1Page (Manage Content -> Classic Pages). The E1Page can now be included as part of a Composed Page.



Now you can have your BOT within the JD Edwards EnterpriseOne interface!

ORACLE CORPORATION

Worldwide Headquarters

500 Oracle Parkway, Redwood Shores, CA 94065 USA

Worldwide Inquiries

TELE + 1.650.506.7000 + 1.800.ORACLE1

FAX + 1.650.506.7200

oracle.com

CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com. Outside North America, find your local office at oracle.com/contact.

 blogs.oracle.com/oracle

 facebook.com/oracle

 twitter.com/oracle

Integrated Cloud Applications & Platform Services

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0918

White Paper: Integrated Chatbot User Experience for the JD Edwards Administration
August 2018



Oracle is committed to developing practices and products that help protect the environment

ORACLE®