WPF Dev Camp - Module 7

Extending Service with Web API

Lab version: 1.0

Last updated: 1/14/2015

Estimated demo delivery time: **20 minutes**.



Contents

[Overview 3](#_Toc409011726)

[Prerequisites 3](#_Toc409011727)

[Setup 3](#_Toc409011728)

[Exercise 1: Using Web API to Create RESTful Services 4](#_Toc409011729)

[Task 1: Using Visual Studio Scaffolding to Prototype a Web API Endpoint 4](#_Toc409011730)

[Task 2: Testing the Scaffolded Web API Endpoint 7](#_Toc409011731)

[Task 3: Customizing and Extending the Scaffolded Web API 9](#_Toc409011732)

[Task 4: Using the Web API with Other Clients 13](#_Toc409011733)

[Task 5: Securing the Web API Endpoint 16](#_Toc409011734)

# Overview

In this demo, we introduce a scenario where the engineering team has been asked to extend the Expenses service so that it is easier to integrate with any number of devices such as phones and tablets using a RESTful interface. In order to accomplish this, the team plans to develop the new Expenses API using ASP.NET Web API, and then roll this out in parallel with the existing WCF service remaining in place for the foreseeable future.

# Prerequisites

The following are required to complete this demo:

* [Microsoft Visual Studio 2013](http://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx) (tested with Update 4)
* [Visual Studio Tools for Apache Cordova](http://go.microsoft.com/fwlink/p/?LinkId=397606) (CTP3.0 shown in this demo)
  + Note: create and run a blank Cordova app using the Android Emulator ahead of time, as the first deployment can take some time.
* [Postman](http://www.getpostman.com/) (can also use browser or other HTTP debugging proxy software of choice)
* Internet connection
* [Microsoft Azure](http://azure.microsoft.com/en-us/pricing/free-trial/) subscription
* Expenses codebase

# Setup

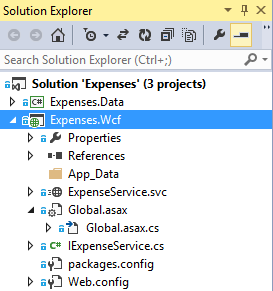
* Load and build the Expenses solution to ensure that it builds correctly (**use the solution as it was left at the end of the demo titled “Using Service Bus Relay to Connect to On-Premises Service”**).
* Load and build the Expenses solution to ensure that it builds correctly.

# Exercise 1: Using Web API to Create RESTful Services

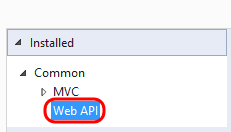
In this exercise, we will demonstrate how to use the Visual Studio Scaffolding feature to kick start a new ASP.NET Web API endpoint, to better allow additional clients to connect to the Expense service. We will then spend a few minutes customizing the Web API endpoint and finally show a proof-of-concept demo using Apache Cordova and the Android Emulator with a client connecting to the new service.

## Task 1: Using Visual Studio Scaffolding to Prototype a Web API Endpoint

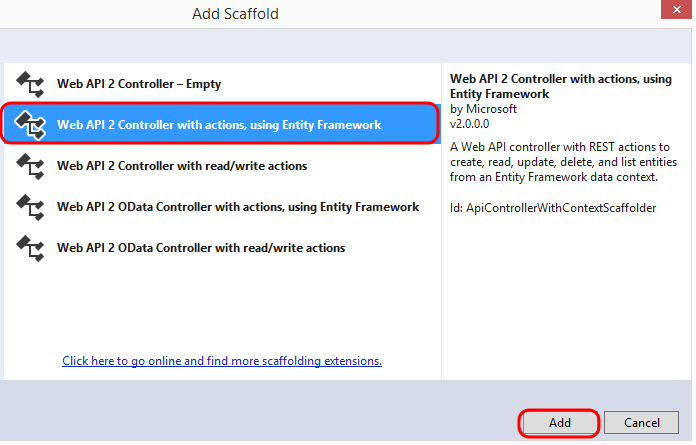
1. Open the Expenses solution in Visual Studio and expand the Expenses.Wcf project node.



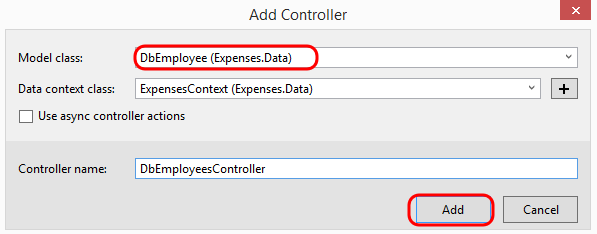
1. Right-click on the Expenses.Wcf project node and then select Add | New Scaffolded Item.
2. The Add Scaffold window shows a few different scaffolding choices in two different categories, including MVC and Web API. Select the **Web API** category option.



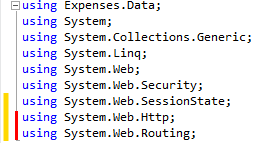
1. Since our data layer uses Entity Framework, go ahead and select the “Web API Controller with actions, using Entity Framework” option and then click Add. Also note that OData is an option here as well.



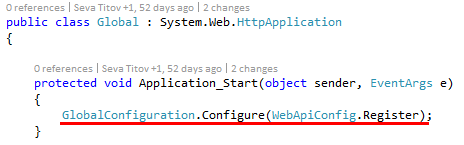
1. For this demonstration, select the DbEmployee model class. The ExpensesContext should be the only option available, and therefore selected by default. Use the default controller name and then click the Add button.



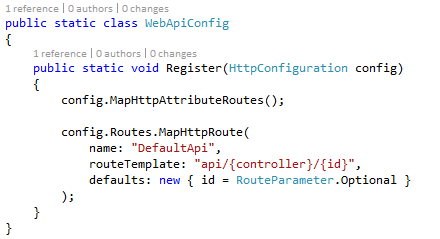
1. After the scaffolding process completes, the project will have some additional references, code, and configuration added that help accelerate the development with Web API. The first thing that you should note is the readme.txt that is loaded in the editor (which is not added to the project itself). This readme file provides some notes that are necessary to fully tie together the existing web application with the new Web API code. Follow the first step of the instructions by copying and pasting the listed namespace references to the top of Global.asax.cs.



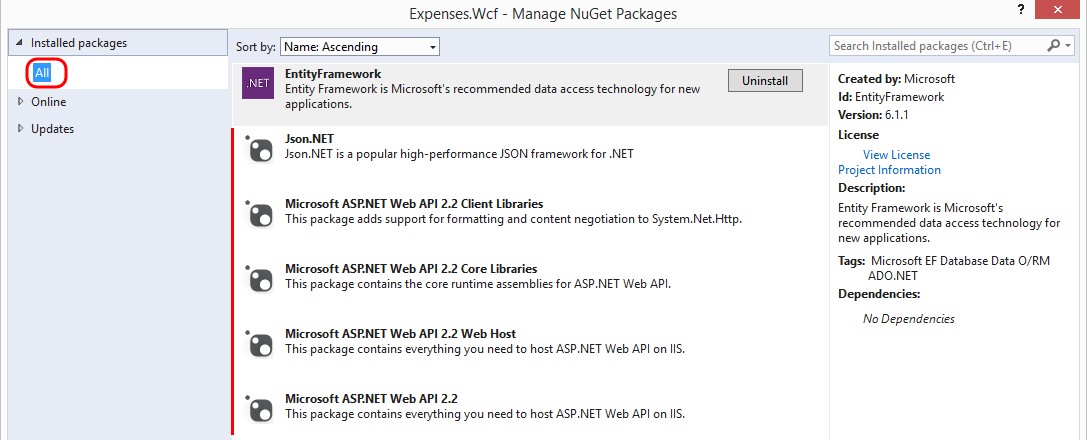
1. Return to readme.txt and note that the second step is to define the Application\_Start method. Since we already have that defined in Global.asax.cs, there is no additional action here.
2. Step three is a configuration step for Web API. Copy and paste the line of code to the body of the Application\_Start method.



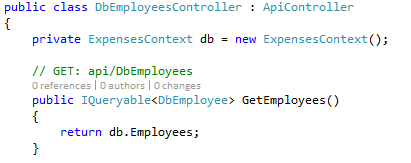
1. Open WebApiConfig.cs in the editor. This static class, which we just saw is called from Application\_Start, configures the routing for our Web API endpoints (including routes defined using attributes in code).



1. Right-click on the Expenses.Wcf project node and select Manage NuGet Packages. Note that the scaffolding wizard also added in the appropriate ASP.NET Web API libraries and one for JSON serialization (you need to select the Installed Packages node).

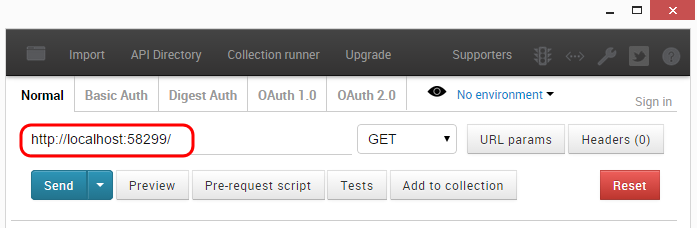


1. Close the Manage NuGet Packages window.
2. Open DbEmployeesController.cs in a code editor window. This class was generated by the scaffolding process and contains a starting point for CRUD operations. The class itself inherits from ApiController.

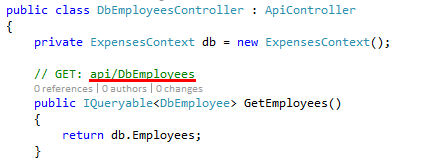


## Task 2: Testing the Scaffolded Web API Endpoint

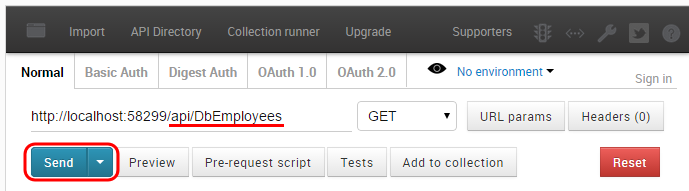
1. Right-click on the solution node in Solution Explorer and select the “Single startup project” option followed by Expenses.Wcf.
2. Press F5 to launch the updated application in IIS Express.
3. Launch Postman and then enter the base URL for the Expenses service application as it runs locally in IIS Express (you can get this by selecting the Expenses.Wcf project node and pressing F4 to show the Properties window).



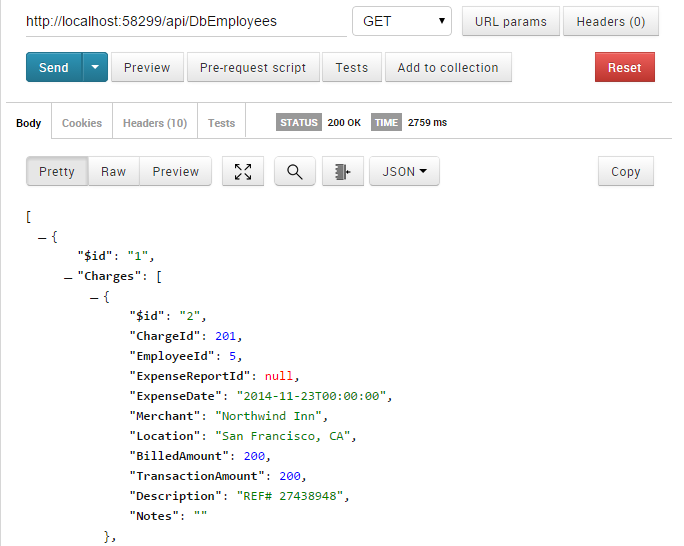
1. Copy the path component from the comment just above the DbEmployeesController.GetEmployees method.



1. Back in Postman, paste the copied path to the end of the base URL that you are building. Click Send to test the call to the new Web API.



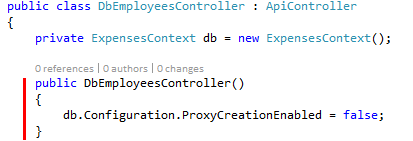
1. Take a look at the JSON results that are returned. This shows that all employees in the database plus there charges were returned.



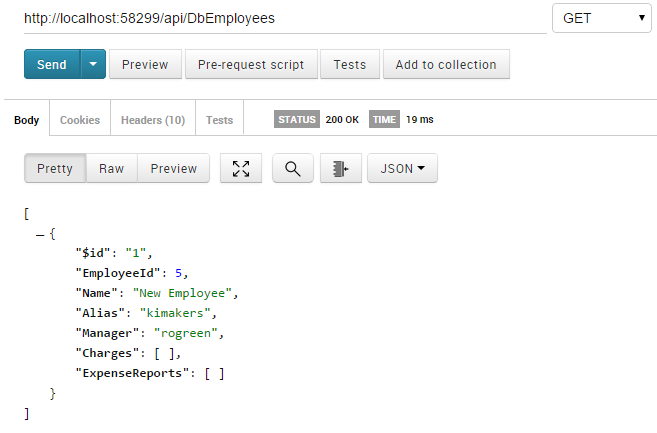
1. Stop debugging in Visual Studio.

## Task 3: Customizing and Extending the Scaffolded Web API

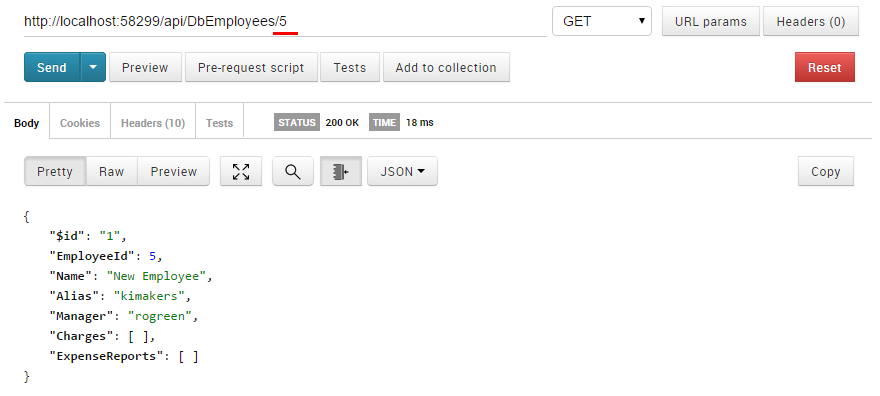
1. Let’s say that we would rather not have all charges returned with the list of employees. An easy fix is to add the following constructor to the controller in order to disable proxy creation:



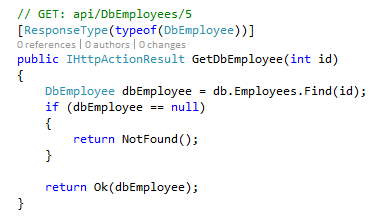
1. Debug the service project once again and click Send in Postman to send the same request as before. Note that this time the Charges and ExpenseReports fields are empty as expected. Also take note of the EmployeeId value (yours may be different than the screenshot below).



1. The scaffolding also created an API to get a specific employee by their ID value. In Postman, add a forward slash plus the EmployeeId value that you determined in the previous step to the URL and then click Send. Only the selected employee is returned.



1. Stop debugging in Visual Studio and return to the controller definition. Locate the method that defines the API which returns an employee based on the ID value passed. This represents a typical Web API definition where you have a ResponseType declaration and a method name that begins with the HTTP request type associated with it (Get, Put, and so on).



1. Let’s say that we would like to add in a similar API that returns the employee based on the alias provided, instead of an ID value. Copy and paste in the following code to the controller:

// GET: api/DbEmployees/alias/kimakers

[ResponseType(typeof(DbEmployee))]

[Route("api/DbEmployees/alias/{alias}")]

public IHttpActionResult GetDbEmployeeByAlias(string alias)

{

DbEmployee dbEmployee = db.Employees.FirstOrDefault(item => item.Alias == alias);

if (dbEmployee == null)

{

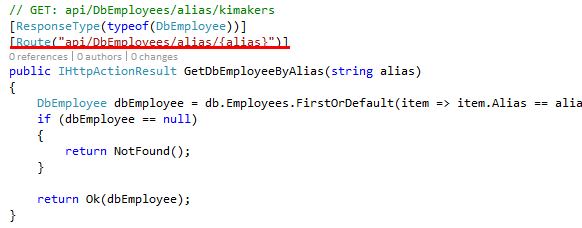
return NotFound();

}

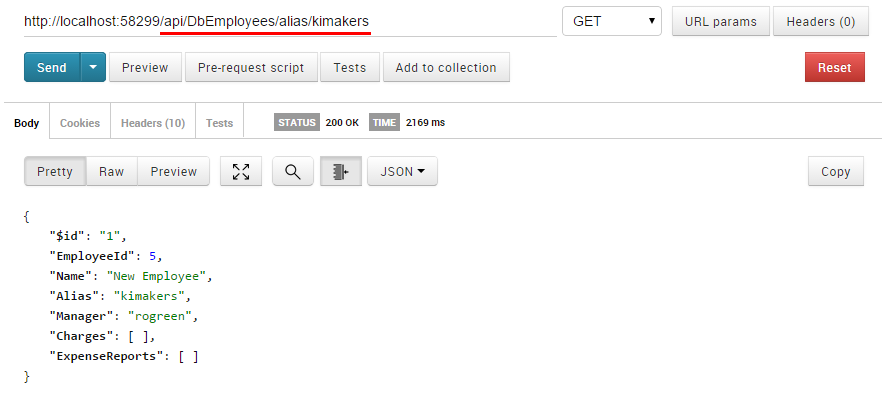
return Ok(dbEmployee);

}

1. Note the use of a routing attribute is necessary here, since the existing GetDbEmployee method is already in place and that would otherwise create a conflict.



1. F5 the service once again, test the new API with a call “/api/DbEmployees/alias/kimakers” in Postman, and then return to Visual Studio and stop debugging.



1. Let’s say that we want to add in a Web API method to return approved expense reports for a given alias. Copy and paste the following code into the controller:

[ResponseType(typeof(DbEmployee))]

[Route("api/DbEmployees/alias/{alias}/approved")]

public IHttpActionResult GetApprovedExpenseReports(string alias)

{

DbEmployee dbEmployee = db.Employees.Include(item => item.ExpenseReports).

FirstOrDefault(item => item.Alias == alias);

if (dbEmployee == null)

{

return NotFound();

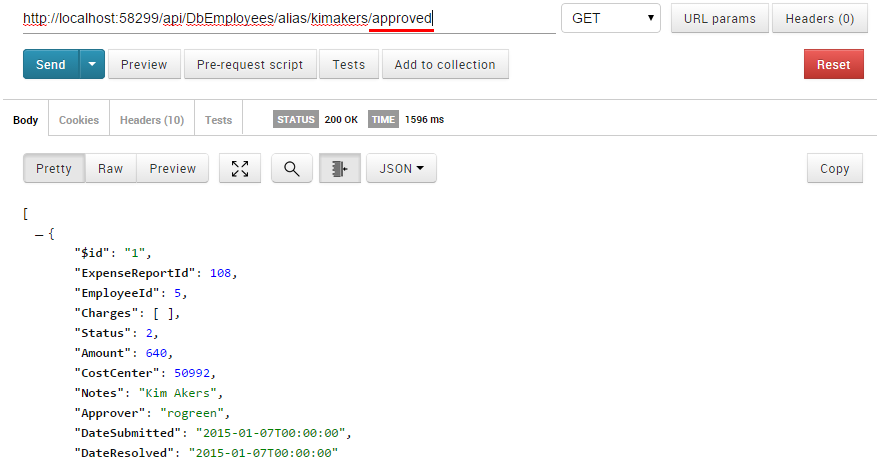
}

return Ok(dbEmployee.ExpenseReports.

Where(item => item.Status == DbExpenseReportStatus.Approved));

}

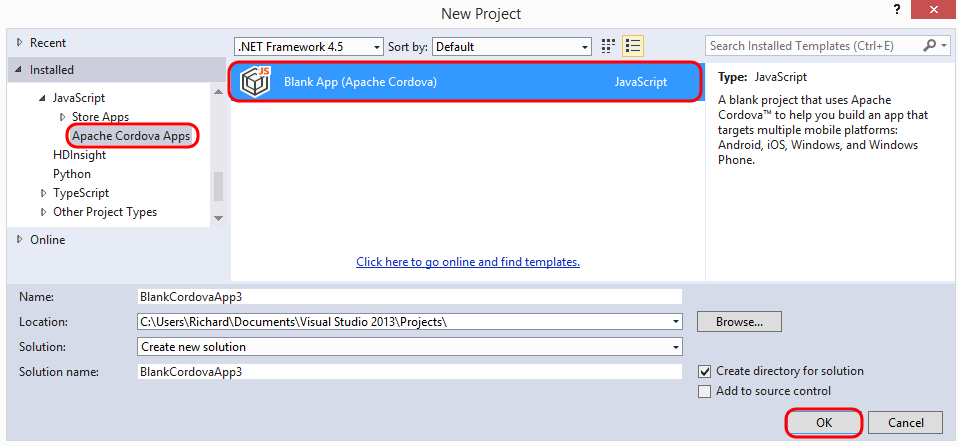
1. F5 the service once again, test the new API with a call “/api/DbEmployees/alias/kimakers/approved” in Postman, and then return to Visual Studio and stop debugging.



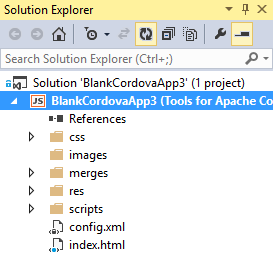
1. Close Visual Studio.

## Task 4: Using the Web API with Other Clients

1. In this task, we will demonstrate how to get started with the creation of a new Apache Cordova application for Android devices using Visual Studio. Launch a new instance of Visual Studio.
2. In the new instance of Visual Studio, select File | New | Project from the main menu.
3. In the New Project window, navigate to the JavaScript | Apache Cordova Apps template category and then select the “Blank App (Apache Cordova)” template. Click OK to accept the default project name and location.



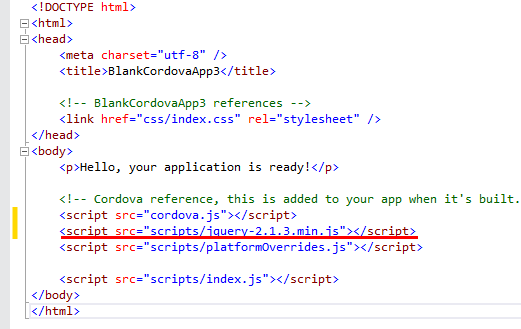
1. Apache Cordova allows developers to create applications using HTML and JavaScript, but at the same time also access native device capabilities. Additional details are beyond the scope of this demonstration, but you can learn more about it on [MSDN](http://www.visualstudio.com/en-us/explore/cordova-vs.aspx) if desired.



1. Select Tools | NuGet Package Manager | Package Manager Console from the main menu.
2. Use the following command grab a copy of jQuery:

Install-Package jQuery -Version 2.1.3

1. Open index.html in the editor and add a Cordova reference to the jQuery script.



1. Open index.js in the editor and add the following code snippet to the end of the onDeviceReady function:

$(document).ready(function () {

$.getJSON("http://**{YOUR AZURE WEBSITE}**/api/DbEmployees/**X** ")

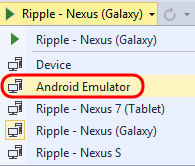
.done(function (employee) {

alert(employee.Alias);

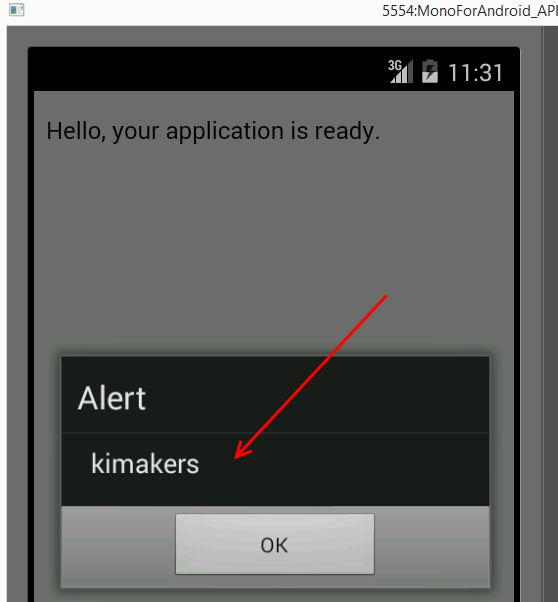
});

});

1. Replace the {YOUR AZURE WEBSITE} with your own, e.g. http://expenses.azurewebsites.net.
2. Replace the ‘X’ with the Employee ID value that you previously determined.
3. This snippet of code will make a request to our Web API when ready to get the employee with a specific ID value (use the same value that you used previously).
4. Select the Android Emulator option from the device selection drop-down in Visual Studio.



1. In the instance of Visual Studio with the Expenses solution open, publish the Expenses.Wcf project to the production website in Azure.
2. After the publication process is successful, return to the instance of Visual Studio with the Apache Cordova app, press F5 to start a debugging session with the Android Emulator. The application should load and you should see an alert with the alias of the user shown.



1. Close the Android emulator and stop debugging in Visual Studio.

## Task 5: Securing the Web API Endpoint

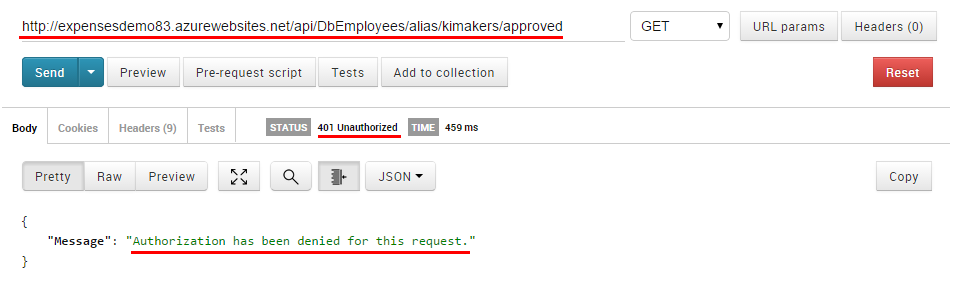
1. Return to the instance of Visual Studio with the Expenses solution open. Web API assumes that authentication happens in the host, in many cases this is IIS. We are not covering authentication in detail in this demo, but we will instead manually authenticate a demo user in order to demonstrate authorization at the Web API controller level.
2. Open DbEmployeesController.cs in the editor and add the following Authorize attribute to the GetApprovedExpenseReports method:

[Authorize(Roles="Administrator")]

1. Deploy the Expenses.Wcf project to the production website in Azure once again.
2. Use Postman to create a test call to get approved expense reports, which will be of the form:

http:// {YOUR AZURE WEBSITE}/api/DbEmployees/alias/kimakers/approved

1. Note that we get back a 401 “Unauthorized” response for the request.



1. Add the following snippet to the Application\_AuthenticateRequest method in Global.asax.cs:

string[] roles = new string[1];

roles[0] = "Administrator";

Thread.CurrentPrincipal = new GenericPrincipal(new GenericIdentity("user", "Password"), roles);

if (HttpContext.Current != null)

{

HttpContext.Current.User = Thread.CurrentPrincipal;

}

1. Add the following using statements to the top of Global.asax.cs:

using System.Threading;

using System.Security.Principal;

1. Publish the secured Web API to the production website in Azure once again.
2. Use Postman to create another test call to get approved expense reports to ensure that everything is now working as expected. Note that we could implement any authentication scheme that we want at this point, from basic authentication to Azure Active Directory. To learn more about authentication and authorization with Web API, please see this [article](http://www.asp.net/web-api/overview/security/authentication-and-authorization-in-aspnet-web-api) on asp.net.