






Add a dashboard widget

03/08/2016 • 30 minutos para ler • Colaboradores      [tudo](#)

Neste artigo

[Preparation and required setup for this tutorial](#)

[What you'll find in the tutorial](#)

[Part 1: Hello World](#)

[Part 2: Hello World with Azure DevOps Services REST API](#)

[Part 3: Hello World with Configuration](#)

Widgets on a dashboard are implemented as [contributions](#) in the [extension framework](#). A single extension can have multiple contributions. In this guide we will show you how to create an extension with multiple widgets as contributions.

This guide is divided into three parts, each building on the previous ones. The goal is to start with a simple widget and end with a comprehensive one.

Preparation and required setup for this tutorial

In order to create extensions for Azure DevOps Services, there are some prerequisite software and tools you'll need:

Knowledge: Some knowledge of JavaScript, HTML, CSS is required for widget development.

- An **organization** in Azure DevOps Services for installing and testing your widget, more information can be found [here](#)
- A **text editor**. For many of the tutorials we used `Visual Studio Code`, which can be downloaded [here](#)
- The latest version of **node**, which can be downloaded [here](#)
- **TFS Cross Platform Command Line Interface (tfx-cli)** to package your extensions.
 - **tfx-cli** can be installed using `npm`, a component of Node.js by running `npm i -g tfx-cli`
- A home directory for your project. This directory will be referred to as `home` throughout the tutorial.

Extension File Structure:

no-highlight

 Copiar

```
|--- README.md
|--- sdk
|   |--- node_modules
|   |--- scripts
|       |--- VSS.SDK.min.js
|--- img
|   |--- logo.png
|--- scripts
|--- hello-world.html           // html page to be used for your widget
|--- vss-extension.json         // extension's manifest
```

What you'll find in the tutorial

1. The first part of this guide shows you how to create a new widget which prints a simple "Hello World"

message.

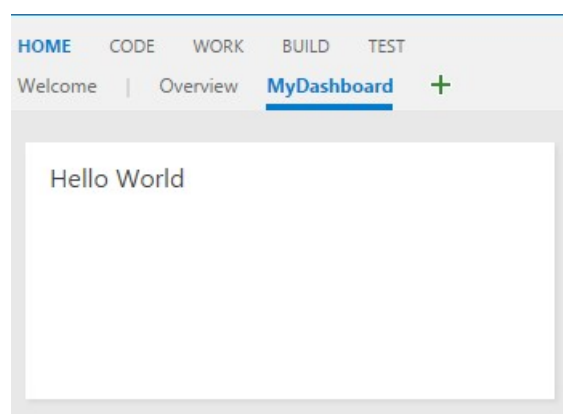
2. The [second part](#) builds on the first one by adding a call to an Azure DevOps Services REST API.
3. The [third part](#) explains how to add configuration to your widget.

If you're in a hurry and want to get your hands on the code right away, you can download the complete samples [here](#). Once downloaded, go to the `widgets` folder, then follow [Step 6](#) and [Step 7](#) directly to publish the sample extension which has the three sample widgets of varying complexities.

Get started with some [basic styles for widgets](#) that we provide out of the box for you and some guidance on widget structure.

Part 1: Hello World

This part presents a widget that prints "Hello World" using JavaScript.



Step 1: Get the client SDK - `vss.SDK.min.js`

The core SDK script, `vss.SDK.min.js`, enables web extensions to communicate to the host Azure DevOps Services frame and to perform operations like initializing, notifying extension is loaded or getting context about the current page. Get the Client SDK `vss.SDK.min.js` file and add it to your web app. Place it in the `home/sdk/scripts` folder.

Use the 'npm install' command to retrieve the SDK:

```
no-highlight
```

[Copiar](#)

```
npm install vss-web-extension-sdk
```

To learn more about the SDK, visit the [Client SDK GitHub Page](#).

Step 2: Your HTML page - `hello-world.html`

This is the glue that holds your layout together and includes references to CSS and JavaScript. You can name this file anything, just be sure to update all references to `hello-world` with the name you use.

Your widget is HTML based and will be hosted in an [iframe](#). Add the below HTML in `hello-world.html`. We add the mandatory reference to `vss.SDK.min.js` file and include an `h2` element in the `body` which will be updated with the string Hello World in the upcoming step.

HTML 

```
<!DOCTYPE html>
<html>
  <head>
    <script src="sdk/scripts/VSS.SDK.min.js"></script>
  </head>
  <body>
    <div class="widget">
      <h2 class="title"></h2>
    </div>
  </body>
</html>
```

Even though we are using an HTML file, most of the HTML head elements other than script and link will be ignored by the framework.

Step 3: Your JavaScript

We use JavaScript to render content in the widget. In this guide we wrap all of our JavaScript code inside a `<script>` element in the HTML file. You can choose to have this in a separate JavaScript file and refer it in the HTML file. Apart from the logic to render the content, this JavaScript code will initialize the VSS SDK, map the code for your widget to your widget name, and notify the extension framework of widget success or failure. In our case, below is the code that would print "Hello World" in the widget. Add this `script` element in the `head` of the HTML.

HTML 

```
<script type="text/javascript">
  VSS.init({
    explicitNotifyLoaded: true,
    usePlatformStyles: true
  });

  VSS.require("TFS/Dashboards/WidgetHelpers", function (WidgetHelpers) {
    WidgetHelpers.IncludeWidgetStyles();
    VSS.register("HelloWorldWidget", function () {
      return {
        load: function (widgetSettings) {
          var $title = $('h2.title');
          $title.text('Hello World');

          return WidgetHelpers.WidgetStatusHelper.Success();
        }
      }
    });
    VSS.notifyLoadSucceeded();
  });
</script>
```

`VSS.init` initializes the handshake between the iframe hosting the widget and the host frame.. We pass `explicitNotifyLoaded: true` so that the widget can explicitly notify the host when we are done loading. This control allows us to notify load completion after ensuring that the dependent modules are loaded. We pass `usePlatformStyles: true` so that the Azure DevOps Services core styles for html elements (such as body, div etc) can be used by the Widget. If the widget prefers to not use these styles, they can pass in `usePlatformStyles: false`.

`VSS.require` is used to load the required VSS script libraries. A call to this method automatically loads general libraries like [jQuery](#) and [jQueryUI](#). In our case we depend on the `WidgetHelpers` library which is used to communicate widget status to the widget framework. Therefore, we pass the corresponding module name `TFS/Dashboards/WidgetHelpers` and a callback to `VSS.require`. The callback is called once the module is loaded. This callback will have the rest of the JavaScript code needed for the widget. At the end of the callback we call `VSS.notifyLoadSucceeded` to notify load completion.

`WidgetHelpers.IncludeWidgetStyles` will include a stylesheet with some [basic css](#) to get you started. Make sure to wrap your content inside a HTML element with class `widget` to make use of these styles.

`VSS.register` is used to map a function in javascript which uniquely identifies the widget among the different contributions in your extension. The name should match the `id` that identifies your contribution as described in [Step 5](#). For widgets, the function that is passed to `VSS.register` should return an object that satisfies the `IWidget` contract, i.e. the returned object should have a `load` property whose value is another function that will have the core logic to render the widget. In our case, it is simply to update the text of the `h2` element to "Hello World". It is this function that is called when the widget framework instantiates your widget. We use the `WidgetStatusHelper` from `WidgetHelpers` to return the `WidgetStatus` as success.

Warning: If this name used to register the widget doesn't match the ID for the contribution in the manifest, then the widget will behave unexpectedly.

The `vss-extension.json` should always be at the root of the folder (in this guide, `HelloWorld`). For all the other files, you can place them in whatever structure you want inside the folder, just make sure to update the references appropriately in the HTML files and in the `vss-extension.json` manifest.

Step 4: Your extension's logo: `logo.png`

Your logo is displayed in the Marketplace, and in the widget catalog once a user installs your extension.

You will need a 98px x 98px catalog icon. Choose an image, name it `logo.png`, and place it in the `img` folder.


To support TFS 2015 Update 3, you will need an additional image that is 330px x 160px. This is a preview image shown in this catalog. Choose an image, name it `preview.png`, and place it in the `img` folder as before.

You can name these images however you want as long as the extension manifest in the next step is updated with the names you use.

Step 5: Your extension's manifest: `vss-extension.json`

- **Every** extension must have an extension manifest file
- Please read the [extension manifest reference](#)
- Find out more about the contribution points in the [extension points reference](#)

Create a json file (`vss-extension.json`, for example) in the `home` directory with the following contents:

JSON	 Copiar
{	

```

    "manifestVersion": 1,
    "id": "vsts-extensions-myExtensions",
    "version": "1.0.0",
    "name": "My First Set of Widgets",
    "description": "Samples containing different widgets extending dashboards",
    "publisher": "fabrikam",
    "targets": [
      {
        "id": "Microsoft.VisualStudio.Services"
      }
    ],
    "icons": {
      "default": "img/logo.png"
    },
    "contributions": [
      {
        "id": "HelloWorldWidget",
        "type": "ms.vss-dashboards-web.widget",
        "targets": [
          "ms.vss-dashboards-web.widget-catalog"
        ],
        "properties": {
          "name": "Hello World Widget",
          "description": "My first widget",
          "catalogImageUrl": "img/CatalogIcon.png",
          "previewImageUrl": "img/preview.png",
          "uri": "hello-world.html",
          "supportedSizes": [
            {
              "rowSpan": 1,
              "columnSpan": 2
            }
          ],
          "supportedScopes": ["project_team"]
        }
      }
    ],
    "files": [
      {
        "path": "hello-world.html", "addressable": true
      },
      {
        "path": "sdk/scripts", "addressable": true
      },
      {
        "path": "img", "addressable": true
      }
    ]
  }

```

ⓘ Observação

The **publisher** here will need to be changed to your publisher name. To create a publisher now, visit [Package/Publish/Install](#).

Icons

The **icons** stanza specifies the path to your extension's icon in your manifest.

Contributions

Each contribution entry defines [certain properties](#).

- The **id** to identify your contribution. This should be unique within an extension. This ID should match with the name you used in [Step 3](#) to register your widget.
- The **type** of contribution. For all widgets, this should be `ms.vss-dashboards-web.widget`.
- The array of **targets** to which the contribution is contributing. For all widgets, this should be `[ms.vss-dashboards-web.widget-catalog]`.
- The **properties** is an object that includes properties for the contribution type. For widgets, the below properties are mandatory.

Property	Description
name	Name of the widget to display in the widget catalog.
description	Description of the widget to display in the widget catalog.
catalogIconUrl	Relative path of the catalog icon that you added in Step 4 to display in the widget catalog. The image should be 98px x 98px. If you have used a different folder structure or a different file name, then this is the place to specify the appropriate relative path.
previewImageUrl	Relative path of the preview image that you added in Step 4 to display in the widget catalog for TFS 2015 Update 3 only. The image should be 330px x 160px. If you have used a different folder structure or a different file name, then this is the place to specify the appropriate relative path.
uri	Relative path of the HTML file that you added in Step 1 . If you have used a different folder structure or a different file name, then this is the place to specify the appropriate relative path.
supportedSizes	Array of sizes supported by your widget. When a widget supports multiple sizes, the first size in the array is the default size of the widget. The <code>widget size</code> is specified in terms of the rows and columns occupied by the widget in the dashboard grid. One row/column corresponds to 160px. Any dimension above 1x1 will get an additional 10px that represent the gutter between widgets. For example, a 3x2 widget will be <code>160*3+10*2</code> wide and <code>160*2+10*1</code> tall. The maximum supported size is <code>4x4</code> .
supportedScopes	At the moment we support only team dashboards. Therefore, the value here has to be <code>project_team</code> . In the future when we support other dashboard scopes, there will be more options to choose from here.

Files

The **files** stanza states the files that you want to include in your package - your HTML page, your scripts, the SDK script and your logo. Set `addressable` to `true` unless you include other files that don't need to be URL-addressable.

ⓘ Observação

For more information about the **extension manifest file**, such as its properties and what they do, check out the [extension manifest reference](#).


Step 6: Package, Publish and Share

Once you've written your extension, the next step towards getting it into the marketplace is to package all of

your files together. All extensions are packaged as VSIX 2.0 compatible .vsix files - Microsoft provides a cross-platform command line interface (CLI) to package your extension.


Get the packaging tool

You can install or update the TFS Cross Platform Command Line Interface (tfx-cli) using `npm`, a component of [Node.js](#), from your command line.

no-highlight	 Copiar
<pre>npm i -g tfx-cli</pre>	

Package your extension

Packaging your extension into a .vsix file is effortless once you have the tfx-cli, simply navigate to your extension's home directory and run the following command.

no-highlight	 Copiar
<pre>tfx extension create --manifest-globs vss-extension.json</pre>	

ⓘ Observação

An extension/integration's version must be incremented on every update.

When updating an existing extension, either update the version in the manifest or pass the

`--rev-version` command line switch. This will increment the *patch* version number of your extension and save the new version to your manifest.

After you have your packaged extension in a .vsix file, you're ready to publish your extension to the marketplace.

Create publisher for the extension

All extensions, including those from Microsoft, are identified as being provided by a publisher. If you aren't already a member of an existing publisher, you'll create one.

1. Sign in to the [Visual Studio Marketplace Publishing Portal](#)
2. If you are not already a member of an existing publisher, you'll be prompted to create a publisher. If you're not prompted to create a publisher, scroll down to the bottom of the page and select *Publish Extensions* underneath **Related Sites**.

- Specify an identifier for your publisher, for example: `mycompany-myteam`
 - This will be used as the value for the `publisher` attribute in your extensions' manifest file.
- Specify a display name for your publisher, for example: `My Team`

3. Review the [Marketplace Publisher Agreement](#) and click **Create**

Now your publisher is defined. In a future release, you'll be able to grant permissions to view and manage your publisher's extensions. This will make it easy (and more secure) for teams and organizations to publish extensions under a common publisher, but without the need to share a set of credentials across a set of users.

You need to update the `vss-extension.json` manifest file in the samples to replace the dummy publisher ID `fabrikam` with your publisher ID.

Publish and share the extension

After creating a publisher, you can now upload your extension to the marketplace.

1. Find the **Upload new extension** button, navigate to your packaged .vsix file, and select *upload*.

You can also upload your extension via the command line by using the `tfx extension publish` command instead of `tfx extension create` to package and publish your extension in one step. You can optionally use `--share-with` to share your extension with one or more accounts after publishing. You'll need a personal access token, too.

no-highlight

 Copiar

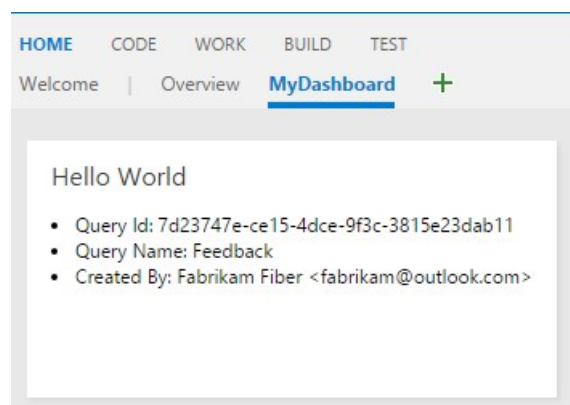
```
tfx extension publish --manifest-globs your-manifest.json --share-with yourOrganization
```

Step 7: Add Widget From the Catalog

Now, go to your team dashboard at <http://dev.azure.com/{yourOrganization}/{yourProject}>. If this page is already open, then refresh it. Hover on the Edit button in the bottom right, and click on the Add button. This should open the widget catalog where you will find the widget you just installed. Choose your widget and click the 'Add' button to add it to your dashboard.

Part 2: Hello World with Azure DevOps Services REST API

Widgets can call any of the [REST APIs](#) in Azure DevOps Services to interact with Azure DevOps Services resources. In this example, we use the REST API for WorkItemTracking to fetch information about an existing query and display some query info in the widget right below the "Hello World" text.



Step 1: HTML

Copy the file `hello-world.html` from the previous example, and rename the copy to `hello-world2.html`. Your folder will now look like below:

no-highlight


 Copiar

```
| --- README.md
| --- sdk
| --- node_modules
```



```
|--- scripts
    |--- VSS.SDK.min.js
|--- img
    |--- logo.png
|--- scripts
|--- hello-world.html           // html page to be used for your widget
|--- hello-world2.html         // renamed copy of hello-world.html
|--- vss-extension.json         // extension's manifest
```


Add a new `div` element right below the `h2` to hold the query information. Update the name of the widget from `HelloWorldWidget` to `HelloWorldWidget2` in the line where you call `VSS.register`. This will allow the framework to uniquely identify the widget within the extension.

HTML	 Copiar
<pre><!DOCTYPE html> <html> <head> <script src="sdk/scripts/VSS.SDK.min.js"></script> <script type="text/javascript"> VSS.init({ explicitNotifyLoaded: true, usePlatformStyles: true }); VSS.require("TFS/Dashboards/WidgetHelpers", function (WidgetHelpers) { WidgetHelpers.IncludeWidgetStyles(); VSS.register("HelloWorldWidget2", function () { return { load: function (widgetSettings) { var \$title = \$('h2.title'); \$title.text('Hello World'); return WidgetHelpers.WidgetStatusHelper.Success(); } } }); VSS.notifyLoadSucceeded(); }); </script> </head> <body> <div class="widget"> <h2 class="title"></h2> <div id="query-info-container"></div> </div> </body> </html></pre>	

Step 2: Access Azure DevOps Services Resources

To enable access to Azure DevOps Services resources, [scopes](#) need to be specified in the extension manifest. We will add the `vso.work` scope to our manifest.

This scope indicates the widget needs read-only access to queries and workitems. See all available scopes [here](#). Add the below at the end of your extension manifest.

JSON	 Copiar
<pre>{ ...,</pre>	

```

    "scopes": [
      "vso.work"
    ]
  }

```

Warning: Adding or changing scopes after an extension is published is currently not supported. If you have already uploaded your extension, you need remove it from the marketplace. Go to **Visual Studio Marketplace Publishing Portal**, right-click on your extension and select "Remove".

Step 3: Make the REST API Call

There are many client-side libraries that can be accessed via the SDK to make REST API calls in Azure DevOps Services. These are called REST clients and are JavaScript wrappers around Ajax calls for all available server side endpoints. You can use methods provided by these clients instead of writing Ajax calls yourself. These methods map the API responses to objects that can be consumed by your code.

In this step, we will update the `VSS.require` call to load `TFS/WorkItemTracking/RestClient` which will provide the WorkItemTracking REST client. We can use this REST client to get information about a query called `Feedback` under the folder `Shared Queries`.

Inside the function that we pass to `VSS.register`, we will create a variable to hold the current project ID. We need this to fetch the query. We will also create a new method `getQueryInfo` to use the REST client. This method that is then called from the load method.

The method `getClient` will give an instance of the REST client we need. The method `getQuery` returns the query wrapped in a promise. The updated `VSS.require` will look as follows:

JavaScript

 Copiar

```

VSS.require(["TFS/Dashboards/WidgetHelpers", "TFS/WorkItemTracking/RestClient"],
  function (WidgetHelpers, TFS_Wit_WebApi) {
    WidgetHelpers.IncludeWidgetStyles();
    VSS.register("HelloWorldWidget2", function () {
      var projectId = VSS.getWebContext().project.id;

      var getQueryInfo = function (widgetSettings) {
        // Get a WIT client to make REST calls to Azure DevOps Services
        return TFS_Wit_WebApi.getClient().getQuery(projectId, "Shared Queries/Feedback")
          .then(function (query) {
            // Do something with the query

            return WidgetHelpers.WidgetStatusHelper.Success();
          }, function (error) {
            return WidgetHelpers.WidgetStatusHelper.Failure(error.message);
          });
      }

      return {
        load: function (widgetSettings) {
          // Set your title
          var $title = $('h2.title');
          $title.text('Hello World');

          return getQueryInfo(widgetSettings);
        }
      };
    });
    VSS.notifyLoadSucceeded();
  }

```

```
});
```

Notice the use of the `Failure` method from `WidgetStatusHelper`. It allows you to indicate to the widget framework that an error has occurred and take advantage to the standard error experience provided to all widgets.

If you do not have the `Feedback` query under the `Shared Queries` folder, then replace `Shared Queries\Feedback` in the code with the path of a query that exists in your project.

Step 4: Display the Response

The last step is to render the query information inside the widget. The `getQuery` function returns an object of type `Contracts.QueryHierarchyItem` inside a promise. In this example, we will display the query ID, the query name, and the name of the query creator under the "Hello World" text. Replace the

`// Do something with the query` comment with the below:

JavaScript

 Copiar

```
// Create a list with query details
var $list = $('<ul>');
$list.append($('<li>').text("Query Id: " + query.id));
$list.append($('<li>').text("Query Name: " + query.name));
$list.append($('<li>').text("Created By: " + ( query.createdBy? query.createdBy.displayName:
"<unknown>" ) ) );

// Append the list to the query-info-container
var $container = $('#query-info-container');
$container.empty();
$container.append($list);
```

Your final `hello-world2.html` will be as follows:

HTML

 Copiar

```
<!DOCTYPE html>
<html>
<head>
  <script src="sdk/scripts/VSS.SDK.min.js"></script>
  <script type="text/javascript">
    VSS.init({
      explicitNotifyLoaded: true,
      usePlatformStyles: true
    });

    VSS.require(["TFS/Dashboards/WidgetHelpers", "TFS/WorkItemTracking/RestClient"],
      function (WidgetHelpers, TFS_Wit_WebApi) {
        WidgetHelpers.IncludeWidgetStyles();
        VSS.register("HelloWorldWidget2", function () {
          var projectId = VSS.getWebContext().projectId;

          var getQueryInfo = function (widgetSettings) {
            // Get a WIT client to make REST calls to Azure DevOps Services
            return TFS_Wit_WebApi.getClient().getQuery(projectId, "Shared
Queries/Feedback")

            .then(function (query) {
              // Create a list with query details
              var $list = $('<ul>');
              $list.append($('<li>').text("Query ID: " + query.id));
              $list.append($('<li>').text("Query Name: " + query.name));
```

```

        $list.append($('li').text("Created By: " + (query.createdBy ?
query.createdBy.displayName: "<unknown>") ));

        // Append the list to the query-info-container
        var $container = $('#query-info-container');
        $container.empty();
        $container.append($list);

        // Use the widget helper and return success as Widget Status
        return WidgetHelpers.WidgetStatusHelper.Success();
    }, function (error) {
        // Use the widget helper and return failure as Widget Status
        return WidgetHelpers.WidgetStatusHelper.Failure(error.message);
    });
}

return {
    load: function (widgetSettings) {
        // Set your title
        var $title = $('h2.title');
        $title.text('Hello World');

        return getQueryInfo(widgetSettings);
    }
}
});
VSS.notifyLoadSucceeded();
});
</script>

</head>
<body>
    <div class="widget">
        <h2 class="title"></h2>
        <div id="query-info-container"></div>
    </div>
</body>
</html>

```

Step 5: Extension Manifest Updates

In this step we will update the extension manifest to include an entry for our second widget. Add a new contribution to the array in the `contributions` property and add the new file `hello-world2.html` to the array in the `files` property. You will need another preview image for the second widget. Name this `preview2.png` and place it in the `img` folder.

JSON

 Copiar

```

{
    ...,
    "contributions": [
        ...,
        {
            "id": "HelloWorldWidget2",
            "type": "ms.vss-dashboards-web.widget",
            "targets": [
                "ms.vss-dashboards-web.widget-catalog"
            ],
            "properties": {
                "name": "Hello World Widget 2 (with API)",
                "description": "My second widget",
                "previewImageUrl": "img/preview2.png",
                "uri": "hello-world2.html",
            }
        }
    ]
}

```

```
        "supportedSizes": [
            {
                "rowSpan": 1,
                "columnSpan": 2
            }
        ],
        "supportedScopes": ["project_team"]
    }
},
"files": [
    {
        "path": "hello-world.html", "addressable": true
    },
    {
        "path": "hello-world2.html", "addressable": true
    },
    {
        "path": "sdk/scripts", "addressable": true
    },
    {
        "path": "img", "addressable": true
    }
],
"scopes": [
    "vso.work"
]
}
```

Step 6: Package, Publish and Share

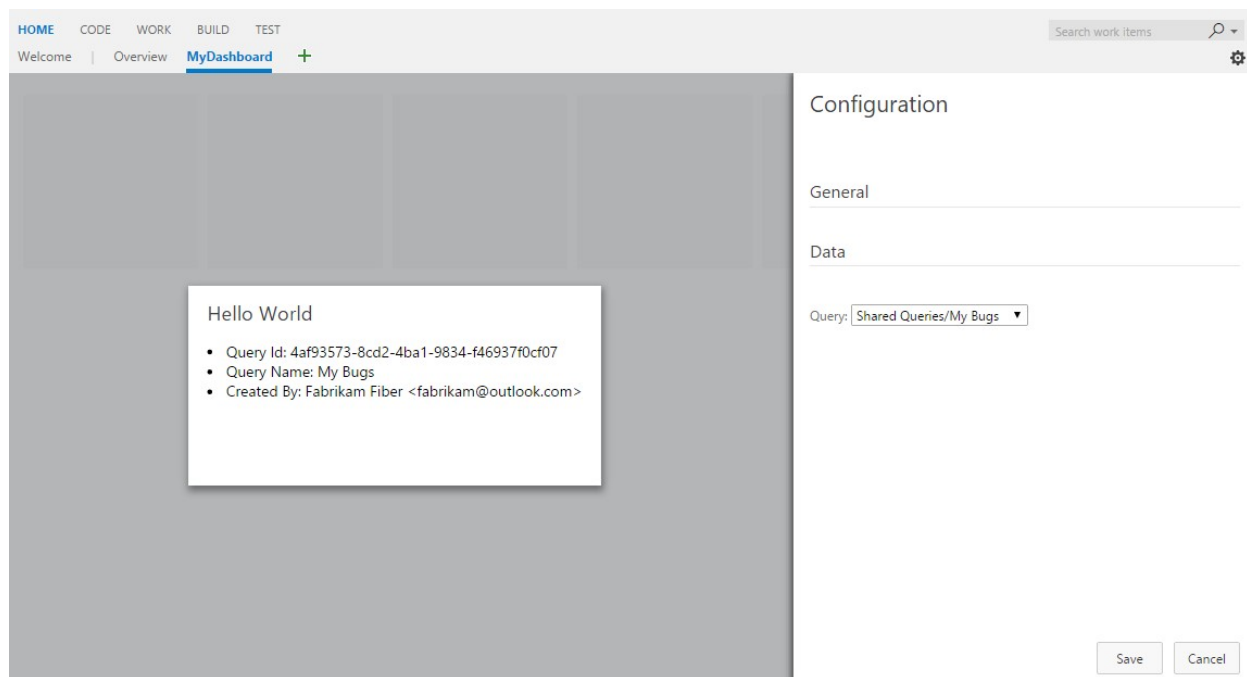
If you have not published your extension yet, then read [this](#) to package, publish and share your extension. If you have already published the extension before this point, you can repackage the extension as described [here](#) and directly [update it](#) to the marketplace.

Step 7: Add Widget From the Catalog

Now, go to your team dashboard at <http://dev.azure.com/{yourOrganization}/{yourProject}>. If this page is already open, then refresh it. Hover on the Edit button in the bottom right, and click on the Add button. This should open the widget catalog where you will find the widget you just installed. Choose your widget and click the 'Add' button to add it to your dashboard.

Part 3: Hello World with Configuration

In [Part 2](#) of this guide, you saw how to create a widget that shows query information for a hard-coded query. In this part, we add the ability to configure the query to be used instead of the hard-coded one. When in configuration mode, the user will get to see a live preview of the widget based on their changes. These changes get saved to the widget on the dashboard when the user clicks the Save button.



Step 1: HTML

Implementations of Widgets and Widget Configurations are a lot alike. Both are implemented in the extension framework as contributions. Both use the same SDK file, `VSS.SDK.min.js`. Both are based on HTML as well as JavaScript and CSS.

Copy the file `html-world2.html` from the previous example and rename the copy to `hello-world3.html`. Add another HTML file called `configuration.html`. Your folder will now look like the below:

no-highlight Copiar

```

|--- README.md
|--- sdk
|   |--- node_modules
|   |--- scripts
|       |--- VSS.SDK.min.js
|--- img
|   |--- logo.png
|--- scripts
|--- configuration.html
|--- hello-world.html           // html page to be used for your widget
|--- hello-world2.html         // renamed copy of hello-world.html
|--- hello-world3.html         // renamed copy of hello-world2.html
|--- vss-extension.json         // extension's manifest

```

Add the below HTML in `configuration.html`. We basically add the mandatory reference to the `VSS.SDK.min.js` file and a `select` element for the dropdown to select a query from a preset list.

HTML Copiar

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="sdk/scripts/VSS.SDK.min.js"></script>
  </head>
  <body>
    <div class="container">

```

```

        <fieldset>
            <label class="label">Query: </label>
            <select id="query-path-dropdown" style="margin-top:10px">
                <option value="" selected disabled hidden>Please select a query</option>
                <option value="Shared Queries/Feedback">Shared Queries/Feedback</option>
                <option value="Shared Queries/My Bugs">Shared Queries/My Bugs</option>
                <option value="Shared Queries/My Tasks">Shared Queries/My Tasks</option>
            </select>
        </fieldset>
    </div>
</body>
</html>

```

Step 2: JavaScript - Configuration

We use Javascript to render content in the widget configuration just like we did for the widget in [Step 3](#) of Part 1 in this guide. Apart from the logic to render the content, this Javascript code will initialize the VSS SDK, map the code for your widget configuration to the configuration name and pass the configuration settings to the framework. In our case, below is the code that loads the widget configuration. Open the file

`configuration.html` and the below `<script>` element to the `<head>`.

HTML

 Copiar

```

<script type="text/javascript">
    VSS.init({
        explicitNotifyLoaded: true,
        usePlatformStyles: true
    });

    VSS.require("TFS/Dashboards/WidgetHelpers", function (WidgetHelpers) {
        VSS.register("HelloWorldWidget.Configuration", function () {
            var $queryDropdown = $("#query-path-dropdown");

            return {
                load: function (widgetSettings, widgetConfigurationContext) {
                    var settings = JSON.parse(widgetSettings.customSettings.data);
                    if (settings && settings.queryPath) {
                        $queryDropdown.val(settings.queryPath);
                    }

                    return WidgetHelpers.WidgetStatusHelper.Success();
                },
                onSave: function() {
                    var customSettings = {
                        data: JSON.stringify({
                            queryPath: $queryDropdown.val()
                        })
                    };
                    return WidgetHelpers.WidgetConfigurationSave.Valid(customSettings);
                }
            };
        });
        VSS.notifyLoadSucceeded();
    });
</script>

```

`VSS.init`, `VSS.require` and `VSS.register` play the same role as they played for the widget as described in [Part 1](#). The only difference is that for widget configurations, the function that is passed to `VSS.register` should return an object that satisfies the `IWidgetConfiguration` contract.

The `load` property of the `IWidgetConfiguration` contract should have a function as its value. This function

will have the set of steps to render the widget configuration. In our case it is simply to update the selected value of the dropdown element with existing settings if any. It is this function that is called when the framework instantiates your `widget configuration`

The `onSave` property of the `IWidgetConfiguration` contract should have a function as its value. This is the function that is called by the framework when user clicks the "Save" button in the configuration pane. If the user input is ready to save, then serialize it to a string, form the `custom settings` object and use `WidgetConfigurationSave.Valid()` to save the user input..

In this guide we use JSON to serialize the user input into a string. You can choose any other way to serialize the user input to string. It will be accessible to the widget via the `customSettings` property of the `WidgetSettings` object. The widget will then have to deserialize this which is covered in [Step 4](#).

Step 3: JavaScript - Enable Live Preview

To enable live preview update when the user selects a query from the dropdown, we attach a change event handler to the button. This handler will notify the framework that the configuration has changed. It will also pass the `customSettings` to be used for updating the preview. To notify the framework, the `notify` method on the `widgetConfigurationContext` needs to be called. It takes two parameters, the name of the event, which in this case is `WidgetHelpers.WidgetEvent.ConfigurationChange`, and an `EventArgs` object for the event, created from the `customSettings` with the help of `WidgetEvent.Args` helper method.

Add the below in the function assigned to the `load` property.

JavaScript

 Copiar

```
$queryDropdown.on("change", function () {
    var customSettings = {
        data: JSON.stringify({
            queryPath: $queryDropdown.val()
        })
    };
    var eventName = WidgetHelpers.WidgetEvent.ConfigurationChange;
    var eventArgs = WidgetHelpers.WidgetEvent.Args(customSettings);
    widgetConfigurationContext.notify(eventName, eventArgs);
});
```

You need to notify the framework of configuration change at least once so that the "Save" button can be enabled.

At the end, your `configuration.html` looks like this:

HTML

 Copiar

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script src="sdk/scripts/VSS.SDK.min.js"></script>
    <script type="text/javascript">
      VSS.init({
        explicitNotifyLoaded: true,
        usePlatformStyles: true
      });

      VSS.require("TFS/Dashboards/WidgetHelpers", function (WidgetHelpers) {
        VSS.register("HelloWorldWidget.Configuration", function () {
```



```

        var $queryDropdown = $("#query-path-dropdown");

        return {
            load: function (widgetSettings, widgetConfigurationContext) {
                var settings = JSON.parse(widgetSettings.customSettings.data);
                if (settings && settings.queryPath) {
                    $queryDropdown.val(settings.queryPath);
                }

                $queryDropdown.on("change", function () {
                    var customSettings = {data: JSON.stringify({queryPath:
$widgetConfigurationContext.notify(eventName, eventArgs);
                });

                return WidgetHelpers.WidgetStatusHelper.Success();
            },
            onSave: function() {
                var customSettings = {data: JSON.stringify({queryPath: $query-
Dropdown.val()});

                return WidgetHelpers.WidgetConfigurationSave.Valid(customSet-
tings);
            }
        });
        VSS.notifyLoadSucceeded();
    });
</script>
</head>
<body>
    <div class="container">
        <fieldset>
            <label class="label">Query: </label>
            <select id="query-path-dropdown" style="margin-top:10px">
                <option value="" selected disabled hidden>Please select a query</option>
                <option value="Shared Queries/Feedback">Shared Queries/Feedback</option>
                <option value="Shared Queries/My Bugs">Shared Queries/My Bugs</option>
                <option value="Shared Queries/My Tasks">Shared Queries/My Tasks</option>
            </select>
        </fieldset>
    </div>
</body>
</html>

```

Step 4: JavaScript - Implement Reload in The Widget

Till this point what we have done is set up widget configuration to store the query path selected by the user. We now have to update the code in the widget to use this stored configuration instead of the hard-coded `Shared Queries/Feedback` from the previous example.


Open the file `hello-world3.html` and update the name of the widget from `HelloWorldWidget2` to `HelloWorldWidget3` in the line where you call `VSS.register`. This will allow the framework to uniquely identify the widget within the extension.

The function mapped to `HelloWorldWidget3` via `VSS.register` currently returns an object that satisfies the `IWidget` contract. Since our widget now needs configuration, this function needs to be updated to return an object that satisfies the `IConfigurableWidget` contract. To do this, update the return statement to include a

property called `reload` as below. The value for this property will be a function that calls the `getQueryInfo` method one more time. This `reload` method gets called by the framework every time the user input changes to show the live preview. This is also called when the configuration is saved.

JavaScript	 Copiar
<pre>return { load: function (widgetSettings) { // Set your title var \$title = \$('h2.title'); \$title.text('Hello World'); return getQueryInfo(widgetSettings); }, reload: function (widgetSettings) { return getQueryInfo(widgetSettings); } }</pre>	

The hard-coded query path in `getQueryInfo` should be replaced with the configured query path which can be extracted from the parameter `widgetSettings` that is passed to the method. Add the below in the very beginning of the `getQueryInfo` method and replace the hard-coded `queryPath` with `settings.queryPath`.


JavaScript	 Copiar
<pre>var settings = JSON.parse(widgetSettings.customSettings.data); if (!settings !settings.queryPath) { var \$container = \$('#query-info-container'); \$container.empty(); \$container.text("Sorry nothing to show, please configure a query path."); return WidgetHelpers.WidgetStatusHelper.Success(); }</pre>	

At this point, your widget is ready to render with the configured settings.

You will notice that both the `load` and the `reload` properties have a similar function. This would be the case for most simple widgets. For complex widgets, there would be certain operations that you would want to run just once no matter how many times the configuration changes. Or there might be some heavy-weight operations that need not run more than once. Such operations would be part of the function corresponding to the `load` property and not the `reload` property.

Step 5: Extension Manifest Updates

Open the `vss-extension.json` file to include two new entries to the array in the `contributions` property. One for the `HelloWorldWidget3` widget and the other for its configuration. You will need yet another preview image for the third widget. Name this `preview3.png` and place it in the `img` folder. Update the array in the `files` property to include the two new HTML files we have added in this example.

JSON	 Copiar
<pre>{ ... "contributions": [... ,</pre>	

```

{
  "id": "HelloWorldWidget3",
  "type": "ms.vss-dashboards-web.widget",
  "targets": [
    "ms.vss-dashboards-web.widget-catalog",
    "fabrikam.vsts-extensions-myExtensions.HelloWorldWidget.Configuration"
  ],
  "properties": {
    "name": "Hello World Widget 3 (with config)",
    "description": "My third widget",
    "previewImageUrl": "img/preview3.png",
    "uri": "hello-world3.html",
    "supportedSizes": [
      {
        "rowSpan": 1,
        "columnSpan": 2
      }
    ],
    "supportedScopes": ["project_team"]
  }
},
{
  "id": "HelloWorldWidget.Configuration",
  "type": "ms.vss-dashboards-web.widget-configuration",
  "targets": [ "ms.vss-dashboards-web.widget-configuration" ],
  "properties": {
    "name": "HelloWorldWidget Configuration",
    "description": "Configures HelloWorldWidget",
    "uri": "configuration.html"
  }
}
],
"files": [
  {
    "path": "hello-world.html", "addressable": true
  },
  {
    "path": "hello-world2.html", "addressable": true
  },
  {
    "path": "hello-world3.html", "addressable": true
  },
  {
    "path": "configuration.html", "addressable": true
  },
  {
    "path": "sdk/scripts", "addressable": true
  },
  {
    "path": "img", "addressable": true
  }
],
...
}

```

Note that the contribution for widget configuration follows a slightly different model than the widget itself. A contribution entry for widget configuration has:

- The **id** to identify your contribution. This should be unique within an extension.
- The **type** of contribution. For all widget configurations, this should be `ms.vss-dashboards-web.widget-configuration`
- The array of **targets** to which the contribution is contributing. For all widget configurations, this will have

a single entry: `ms.vss-dashboards-web.widget-configuration` .

- The **properties** that contains a set of properties which includes name, description, and the URI of the HTML file used for configuration.

To support configuration, the widget contribution needs to be changed as well. The array of **targets** for the widget needs to be updated to include the ID for the configuration in the form `< publisher >.<`

`id for the extension >.< id for the configuration contribution >` which in this case will be

`fabrikam.vsts-extensions-myExtensions.HelloWorldWidget.Configuration`

Warning: If the contribution entry for your configurable widget does not target the configuration using the right publisher and extension name as described above, then configure button will not show up for the widget.

At the end of this part, the manifest file should contains three widgets and one configuration. You can get the complete manifest from the sample [here](#).

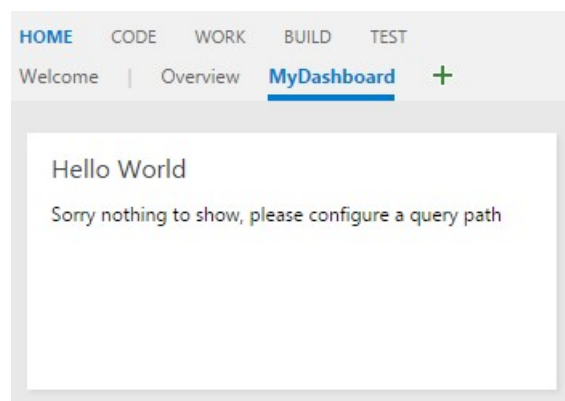
Step 6: Package, Publish and Share

If you have not published your extension yet, then read [this](#) to package, publish and share your extension. If you have already published the extension before this point, you can repackage the extension as described [here](#) and directly [update it](#) to the marketplace.

Step 7: Add Widget From the Catalog

Now, go to your team dashboard at `http://dev.azure.com/{yourOrganization}/{yourProject}`. If this page is already open, refresh it. Hover on the Edit button in the bottom right, and click on the Add button. This should open the widget catalog where you will find the widget you just installed. Choose your widget and click the 'Add' button to add it to your dashboard.

You would see a message asking you to configure the widget.



There are 2 ways to configure widgets. One is to hover on the widget, click on the ellipsis that appears on the top right corner and then click Configure. The other is to click on the Edit button in the bottom right of the dashboard, and then click the configure button that appears on the top right corner of the widget. Either will open the configuration experience on the right side, and a preview of your widget in the center. Go ahead and choose a query from the dropdown. The live preview will show the updated results. Click on "Save" and your widget will display the updated results.

Step 8: Configure More (optional)

You can add as many HTML form elements as you need in the `configuration.html` for additional configuration. There are two configurable features that are available out of the box: widget name and widget size.

By default, the name that you provide for your widget in the extension manifest is stored as the widget name for every instance of your widget that ever gets added to a dashboard. You can allow users to configure this, so that they can add any name they want to their instance of your widget. To allow such configuration, add `isNameConfigurable: true` in the properties section for your widget in the extension manifest.

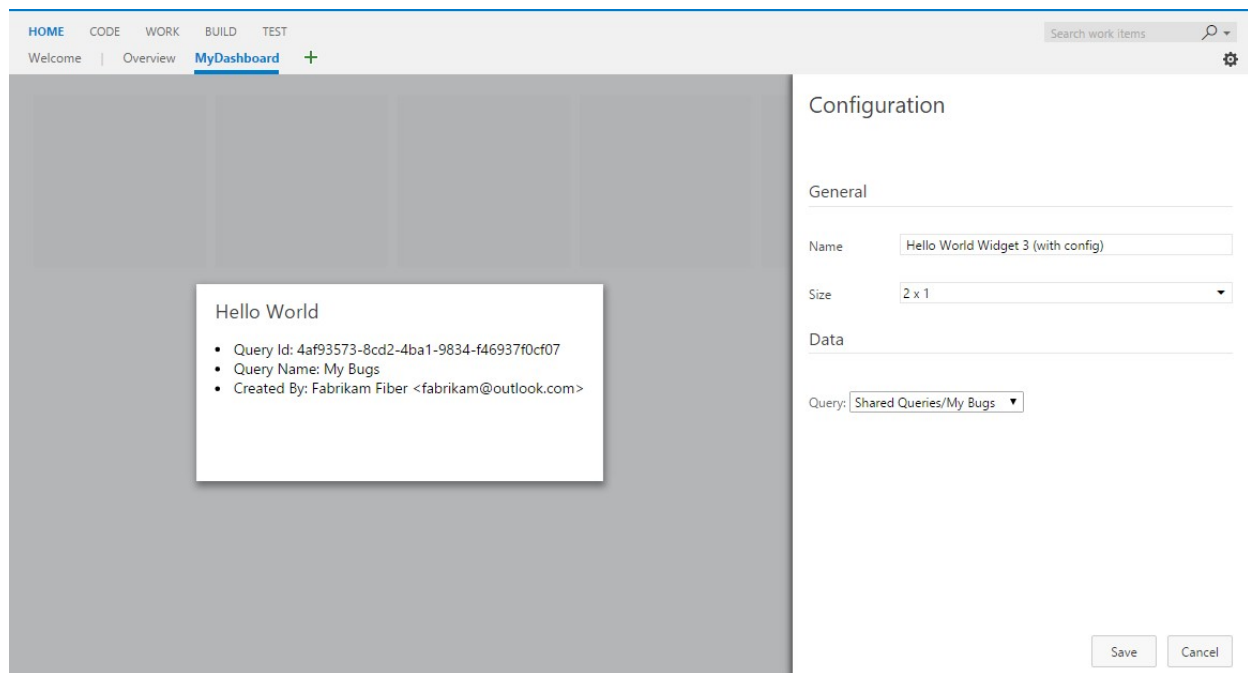
If you provide more than one entry for your widget in the `supportedSizes` array in the extension manifest, then users can configure the widget's size as well.

The extension manifest for the third sample in this guide would look like the below if we enable the widget name and size configuration:

JSON Copiar

```
{
  ...
  "contributions": [
    ... ,
    {
      "id": "HelloWorldWidget3",
      "type": "ms.vss-dashboards-web.widget",
      "targets": [
        "ms.vss-dashboards-web.widget-catalog", "fabrikam.vsts-extensions-
myExtensions.HelloWorldWidget.Configuration"
      ],
      "properties": {
        "name": "Hello World Widget 3 (with config)",
        "description": "My third widget",
        "previewImageUrl": "img/preview3.png",
        "uri": "hello-world3.html",
        "isNameConfigurable": true,
        "supportedSizes": [
          {
            "rowSpan": 1,
            "columnSpan": 2
          },
          {
            "rowSpan": 2,
            "columnSpan": 2
          }
        ],
        "supportedScopes": ["project_team"]
      }
    },
    ...
  ]
}
```

With the above change, [repackage](#) and [update](#) your extension. Refresh the dashboard that has this widget (Hello World Widget 3 (with config)). Open the configuration mode for your widget, you should now be able to see the option to change the widget name and size.



Go ahead and choose a different size from the drop down. You will see the live preview get resized. Save the change and the widget on the dashboard will be resized as well.

Warning: If you remove an already supported size, then the widget will fail to load properly. We are working on a fix for a future release.

You will notice that changing the name of the widget does not result in any visible change in the widget. This is because our sample widgets do not display the widget name anywhere. Let us modify the sample code to display the widget name instead of the hard-coded text "Hello World".

To do this, replace the hard-coded text "Hello World" with `widgetSettings.name` in the line where we set the text of the `h2` element. This will ensure that the widget name gets displayed every time the widget gets loaded on page refresh. Since we want the live preview to be updated every time the configuration changes, we should add the same code in the `reload` part of our code as well. The final return statement in `hello-world3.html` will be as follows:

JavaScript Copiar

```
return {
  load: function (widgetSettings) {
    // Set your title
    var $title = $('h2.title');
    $title.text(widgetSettings.name);

    return getQueryInfo(widgetSettings);
  },
  reload: function (widgetSettings) {
    // Set your title
    var $title = $('h2.title');
    $title.text(widgetSettings.name);

    return getQueryInfo(widgetSettings);
  }
}
```

[Repackage](#) and [update](#) your extension again. Refresh the dashboard that has this widget. Any changes to the

widget name in the configuration mode will update the widget title now.