Welcome to Reveal SDK

Here you will find technical information on the Reveal SDK and API methods, including architecture information.

Where to start:

Reveal SDK Overview. Introduction to the SDK main concepts and features.

Installation and Requirements. How to install and what's required.

Desktop SDK Overview. Introduction to the Desktop SDK main concepts and architecture.

- Setup and configuration. Steps required to get the Desktop SDK ready to be used.

WEB .NET SDK Overview. Introduction to the Web .NET SDK main concepts and architecture.

- Setup and configuration. Steps required to get the Web .NET SDK ready to be used.
- Creating your First App. Walkthrough that guides you through the initial steps of showing a dashboard on your web page/application for the first time.

WEB JAVA SDK Overview. Introduction to the Web JAVA SDK main concepts and architecture.

- Setup and configuration. Steps required to get the Web JAVA SDK ready to be used.
- Running the UpMedia Samples. Walkthroughs that guide you run the UpMedia samples provided.

Reveal SDK Overview

- **Desktop SDK** - Reveal Desktop SDK allows you to embed Reveal inside an external (host) Windows application (WPF or WinForms).

- **Web SDK** - Reveal Web SDK allows you to embed Reveal inside an external (host) web application. The SDK for embedding the Reveal web viewer includes two components:

  - Reveal Web Client SDK,
  - Reveal Web Server SDK, supported in two different platforms (.NET and JAVA)

When installing Reveal's SDK, you install the .NET Web SDK and Desktop SDK at the same time. The JAVA SDK is distributed as a set of Maven modules.

Main Features

With Reveal SDK, developers can embed Reveal into their applications. And dashboards can be displayed and even modified by end users.

Reveal SDK can be used to integrate Reveal into applications developed in multiple platforms and technologies: Web, Windows WPF, and Windows Forms.

The containing app can use Reveal SDK to:

- Provide in-memory data to dashboards. If data is already loaded in the containing app there’s no need to store it before opening the dashboard, Reveal can use the built-in InMemory data provider to use that data as the input for the dashboard.
- Configure the dashboard before it gets rendered, for example changing the name of the database or table to use to get the data based on the current user.
- Change dashboard filters before the dashboard gets rendered or even while it’s visible. The containing app can use this feature to synchronize filters or selections in the app with the data visualized in the dashboard.
- Get notified when a data point is selected in the dashboard (like a bar in a chart is clicked), for example to display additional information or trigger an action in the app like navigating to a page with more details.
- And many other features...

System Requirements and Installation

Desktop SDK Requirements

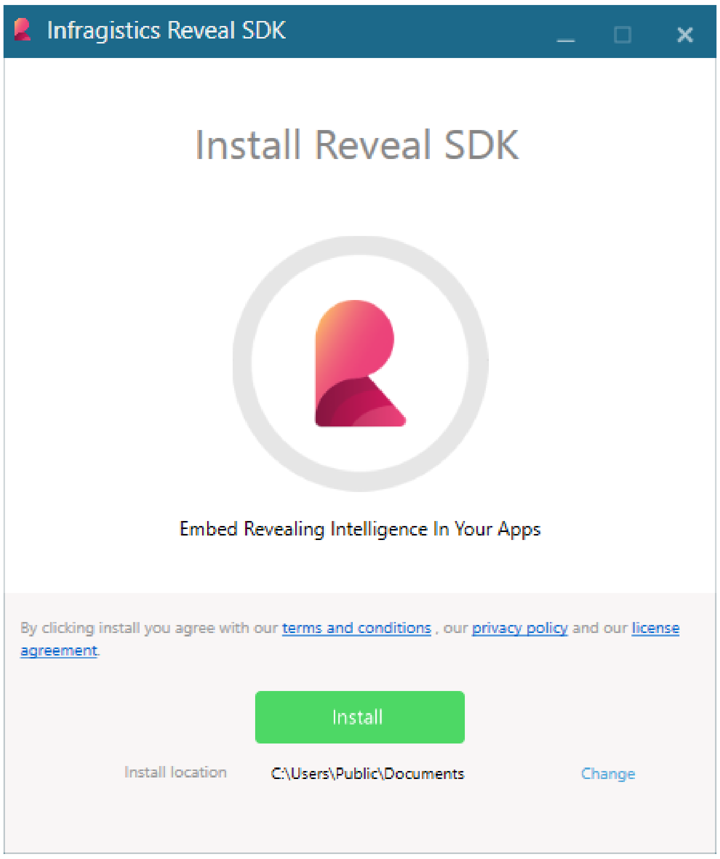
- The Reveal SDK requires .NET version 4.6.2+ and Visual Studio 2015 or up.

Web SDK .NET Requirements

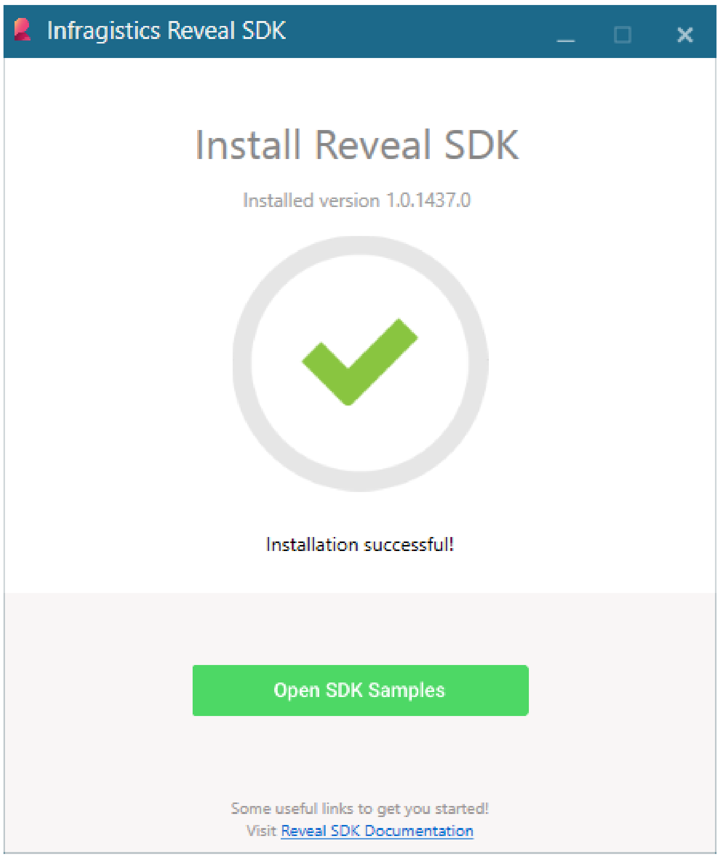
- The Reveal Server SDK requires .NET Core 2.2+ server-side projects targeting .NET framework 4.6.2+.

Installing Desktop and Web .NET SDK

To get the Reveal SDK for both Web and Desktop .NET platforms, sign up here. Once ready, follow through the provided installer:


Infragistics Reveal SDK
Install Reveal SDK
Embed Revealing Intelligence In Your Apps
By clicking install you agree with our terms and conditions , our privacy policy and our license agreement.
Install
Install location
C:\Users\Public\Documents
Change

After a successful installation, you can browse the installed samples by clicking the *Open SDK Samples* link.


Infragistics Reveal SDK
Install Reveal SDK
Installed version 1.0.1437.0
Installation successful!
Open SDK Samples
Some useful links to get you started!
Visit Reveal SDK Documentation

Samples

In case you missed the samples link, you can find them in “%public%\Documents\Infragistics\Reveal\SDK\”.

In this location you will find a solution file (Reveal.Sdk.Samples.sln). This project combines all Web, WPF, and WinForms samples.

For Web you need to restore the node packages in order to run the samples with IIS and change the StartUp project. To restore, just right click the solution in the Solution Explorer and select Restore packages.

Web SDK JAVA Requirements

- Java SDK 11.0.10 and up recommended.
- Maven 3.6.3 and up recommended

Installing JAVA SDK

Reveal Java SDK is distributed as a set of Maven modules. To work with the SDK libraries, you need to add a reference to Reveal's Maven Repository and also a dependency in your Maven pom.xml file. For further details, please refer to Setup and Configuration.

Samples

The **UpMedia samples** illustrate how to use the JAVA SDK, you can get them from GitHub here.

For details about how to run the UpMedia samples, please follow this link.

Getting Dashboards for the SDK

Introduction

Reveal is a business intelligence platform that was purposely designed to be embedded into applications. Embedded Reveal dashboards are a quick and simple way to display information to communicate the status, metrics, or performance of a business.

Let's take a look at the differences between Reveal SDK and the Reveal app.

The **Reveal Application** is a self-service business intelligence tool that enables you to make data-driven decisions faster. You can create, view and share dashboards in your workspaces. And it offers you an identical experience no matter what platform you are on: Web, Desktop, iOS, or Android. For further details about the Reveal app, you can access an **online demo** or browse the **Help Documentation**.

With **Reveal SDK** developers can embed Reveal into their applications developed in multiple platforms and technologies. And Reveal dashboards can be displayed and even modified by end users.

Overview

In order to display dashboards in your application you first need to create them using the Reveal Application on your preferred platform.

You can also use the Reveal SDK to create dashboards from scratch, but the recommended approach when you are first evaluating the SDK is to start with dashboards created with the app.

Getting your Dashboard File

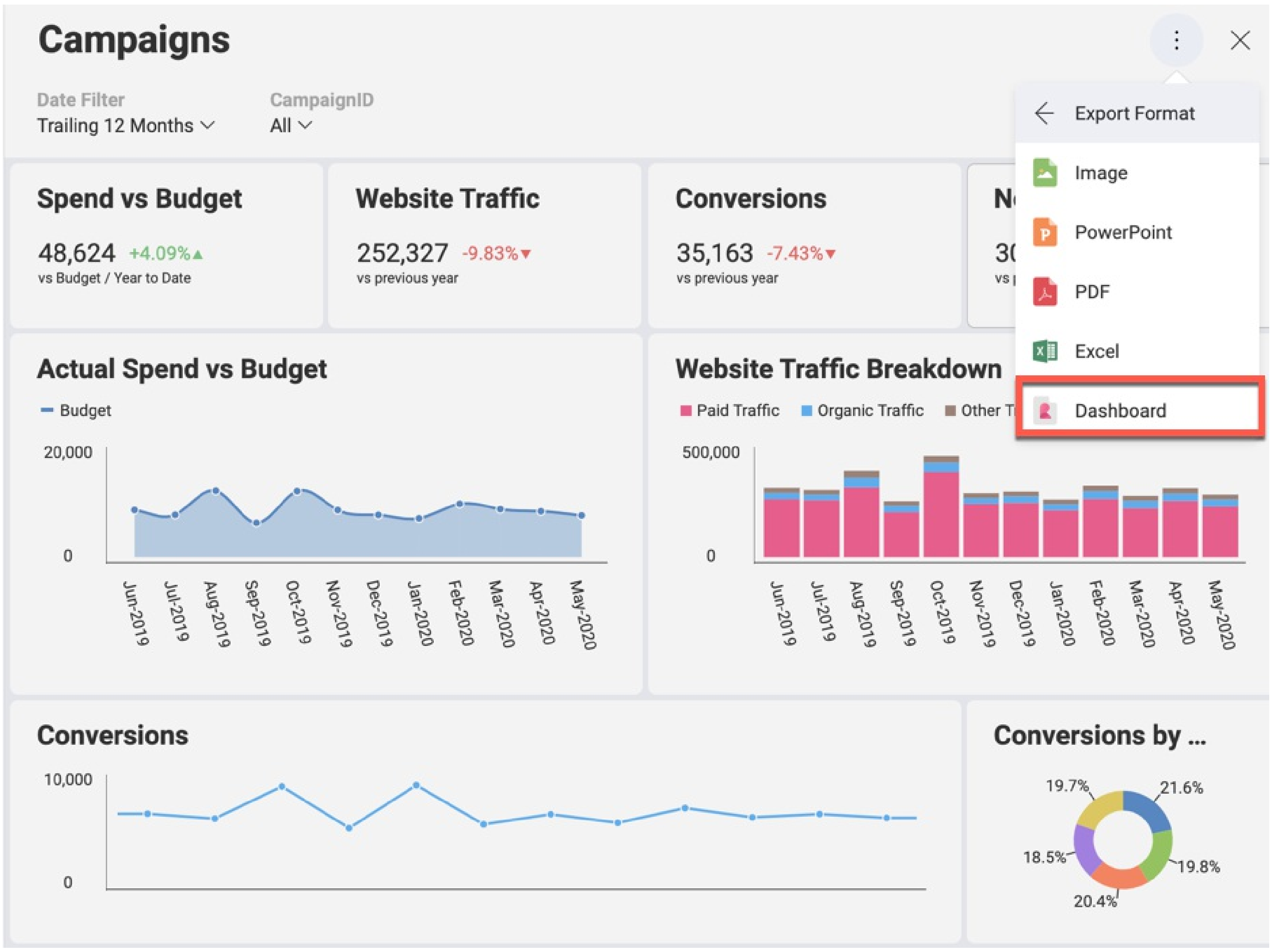
Once you **created your dashboard**, below you can find how to get it:

1. **Open your Dashboard in the Reveal app**

   After installing the Reveal Application in any platform, you can either create your own dashboard or use one of the sample dashboards provided with the app.

2. **Access the Export Options**

   Go to the overflow menu, select *Export*, then *Dashboard*, this will generate a file with extension ".rdash" that you will use later in your application when integrating the SDK.



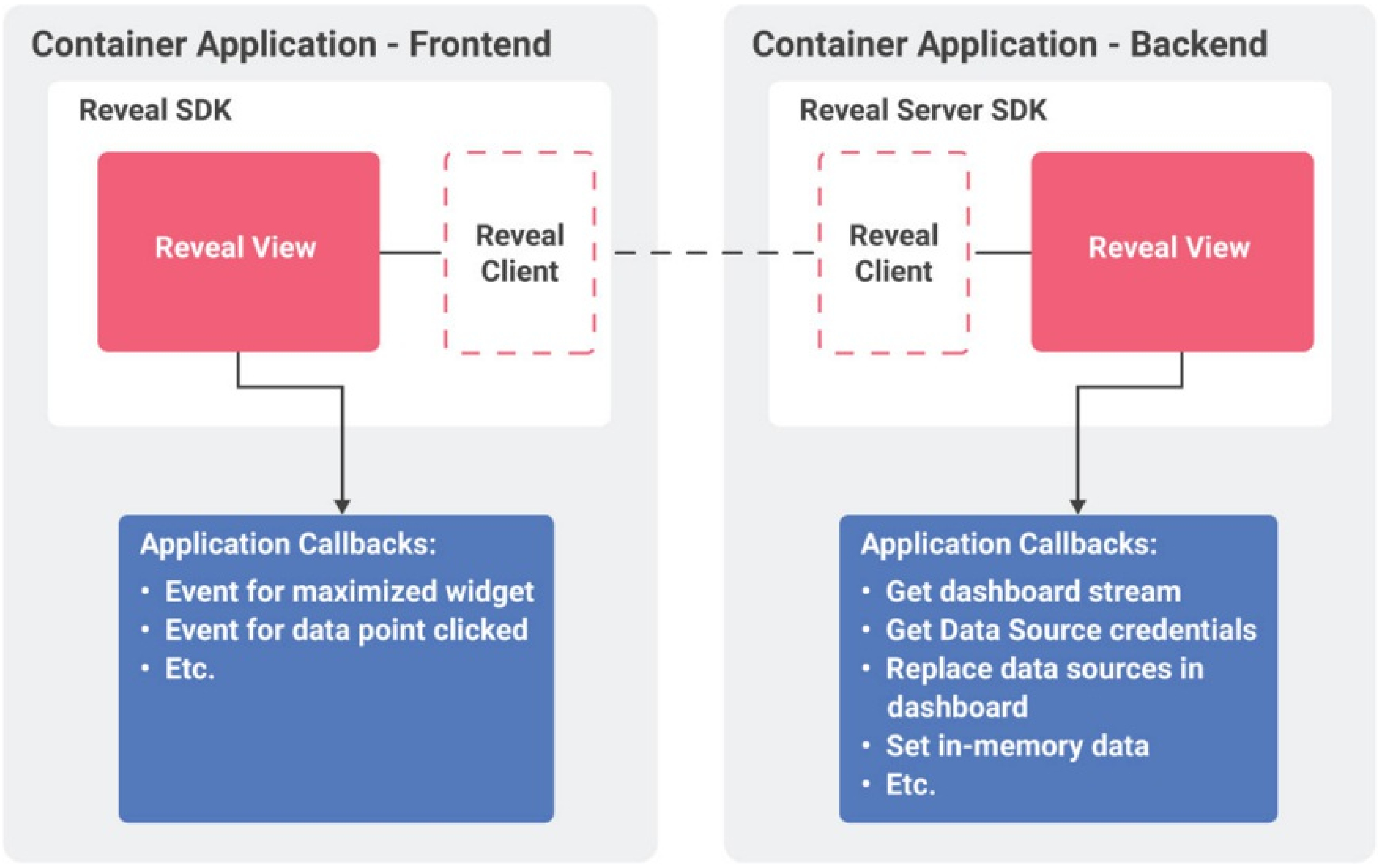
You can export the dashboard file through email (Android and iOS), or as a .rdash file on your computer (Desktop and Web).

Overview

When embedding Reveal into web applications, the architecture is slightly more complex than with native apps, as two components are always involved:

- **Reveal Client SDK**: a set of JavaScript libraries that needs to be integrated into the web application. The frameworks supported today are: jQuery, Angular and React.
- **Reveal Server SDK**: the server-side component to be integrated into the server application, currently this is an ASP.NET Core application targeting .NET Runtime (v4.6.2 or later) and .NET Core (2.2, 3.1, and 5.0).

In the following diagram you visualize the architecture for a web application embedding Reveal Web SDK:


Container Application - Frontend
Reveal SDK
Reveal View
Reveal Client
Application Callbacks:
• Event for maximized widget
• Event for data point clicked
• Etc.
Container Application - Backend
Reveal Server SDK
Reveal Client
Reveal View
Application Callbacks:
• Get dashboard stream
• Get Data Source credentials
• Replace data sources in dashboard
• Set in-memory data
• Etc.

As shown above, the SDK works pretty much the same way as with native apps. The difference is that some of the callbacks are invoked in the client side (like the event sent when a data point is clicked) and others are invoked server side (like the callback to load the dashboard or to provide in-memory data).

Hosting the Client-side and Server-Side Parts on Different Servers

You can host the client-side and the server-side parts separately i.e. on different urls.

To achieve this, set a property on the window object, as shown below:

```
$.ig.RevealSdkSettings.setBaseUrl("{back-end base url}");
```

Please, note that the **trailing slash symbol is required in the URL** in order to set the property successfully.

Set this property **prior to the** *instantiation of the $.ig.RevealView*.

Setup and Configuration

Prerequisites

The Reveal Server SDK requires .NET Core 2.2+ or .NET Framework 4.6.2 ASP MVC application projects.

In case you are targeting .NET Framework 4.6.2+, the Reveal Server SDK supports a win7-x64 runtime environment. To debug your web project you need to add a win7-x64 compatible *RuntimeIdentifier* platform:

```
<PropertyGroup>

  <TargetFramework>net462</TargetFramework>

  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>

</PropertyGroup>
```

Setup and Configuration (Server)

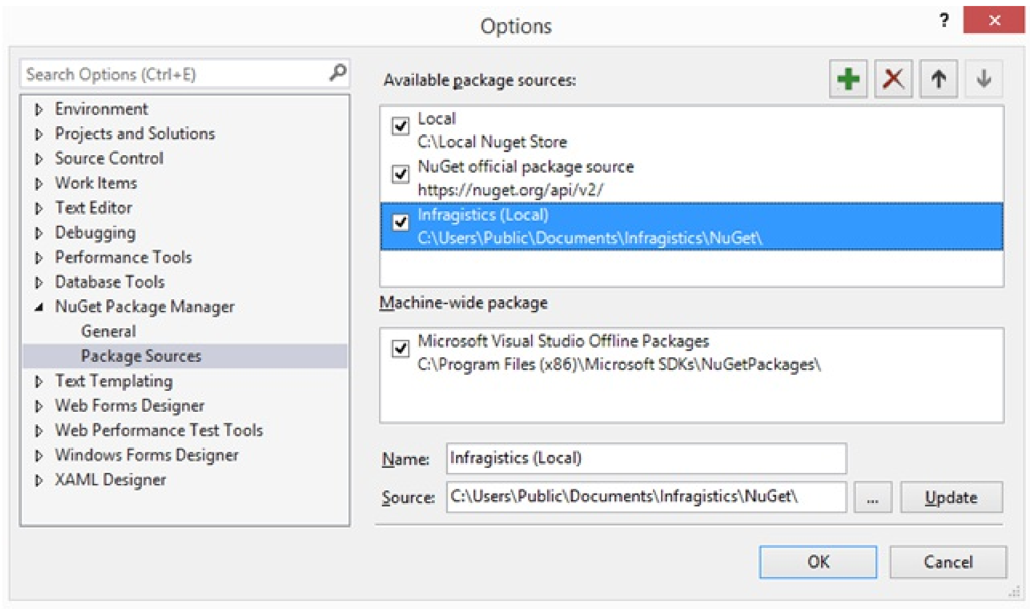
To set up the Reveal Web Server SDK you need to:

1. **Add references to assemblies and install dependency packages.**
2. **Define the Server Context.**
3. **Initialize the Server SDK.**
4. **Enable server-side screenshot generation**.

1. Getting Assemblies and Dependency Packages ready

To add references to assemblies and install dependency packages we recommend using **NuGet** package manager. The easiest way to setup your project is installing **Reveal.Sdk.Web.AspNetCore** (Trial) NuGet package.

After installing the Reveal SDK, you should be able to find a new NuGet package source added to your **nuget.config** called *Infragistics (Local)* that points to “%public%\Documents\Infragistics\NuGet”.



Available package sources:
- [✓] Local C:\Local Nuget Store
- [✓] NuGet official package source https://nuget.org/api/v2/
- [✓] Infragistics (Local) C:\Users\Public\Documents\Infragistics\NuGet\

Machine-wide package
- [✓] Microsoft Visual Studio Offline Packages C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

Name: Infragistics (Local)
Source: C:\Users\Public\Documents\Infragistics\NuGet\

After ensuring you have the Infragistics (Local) feed properly configured by the installer, you need to:

- install the **Reveal.Sdk.Web.AspNetCore** NuGet package to your application project.

- add a NuGet package reference to System.Data.SQLite version 1.0.111+

If you are having issues with the build, follow this **link**.

2. Defining the Server Context

After referencing the required DLLs, you need to create a class that inherits the **RevealSdkContextBase** abstract class. This class allows the Reveal SDK to run inside of your host application and provides callbacks for working with the SDK.

```
using Reveal.Sdk;
public class RevealSdkContext : RevealSdkContextBase
{
    public override IRVDataSourceProvider DataSourceProvider => null;

    public override IRVDataProvider DataProvider => null;

    public override IRVAuthenticationProvider AuthenticationProvider => null;

    public override Task<Dashboard> GetDashboardAsync(string dashboardId)
    {
        var fileName = $"C:\\Temp\\{dashboardId}.rdash";
        var fileStream = new FileStream(fileName, FileMode.Open, FileAccess.Read);
        return Task.FromResult(new Dashboard(fileStream));
    }

    //This callback is used only when "onSave" event is not installed on the
    //RevealView object client side. For more information see the web client SDK documentation
    public override Task SaveDashboardAsync(string userId, string dashboardId, Dashboard dashboard)
    {
        return Task.CompletedTask;
    }
}
```

The implementation above will load dashboards from "C:\Temp" folder, looking for a .rdash file that depends on the *dashboardId* variable. In your application, you may want to change this to load dashboards from another directory, from the database, or even from an embedded resource.

Note

> **Properties returning null:** The first three properties, *DataSourceProvider*, *DataProvider*, and *AuthenticationProvider*, are all implemented to return null. In this guide you can find information about how to implement each of the interfaces for these properties, so they will no longer be implemented to return null.

3. Initializing the Server SDK

In the **Startup.cs**, in the **ConfigureServices** method of the application, call the services extension method *AddRevealServices*, passing in the *RevealEmbedSettings* class.

The *AddRevealServices* extension method is defined in the **Reveal.Sdk** namespace, so you will need to add a using directive. In addition, you also need to set the **CachePath** property as shown below.

```
services.AddRevealServices(new RevealEmbedSettings
{
    LocalFileStoragePath = @"C:\Temp\Reveal\DataSources",
    CachePath = @"C:\Temp"
}, new RevealSdkContext());
```

Note

> **LocalFileStoragePath** is only required if you are using local Excel or CSV files as dashboard data source, and the *RevealSdkContext* class inherits *RevealSdkContextBase* as described above.

Finally, you need to add Reveal endpoints by calling the **AddReveal** extension method when adding MVC service. Similar to the following code snippet:

```
services.AddMvc().AddReveal();
```

Like *AddRevealServices*, the *AddReveal* method is defined in the *Reveal.Sdk* namespace, so you need a using directive too.

4. Enabling server-side screenshot generation

In order to use the **export to image** functionality (either programmatically or through user interaction), you need to perform the steps below:

1. Get the following three files from **<InstallationDirectory>\SDK\Web\JS\Server**:

    - package.json
    - packages-lock.json
    - screenshoteer.js

2. Copy the files to the root level of your project (parent folder of "wwwroot").

3. Make sure you have **npm** (the package manager for Node.js) installed.

If **you don't need the export to image** functionality, you don't need to copy the files to your projects. However, when trying to build the project, it will fail complaining that it cannot find **npm**.

To solve this error, add the following property to your project:

```
<PropertyGroup>
  <DisableRevealExportToImage>true</DisableRevealExportToImage>
</PropertyGroup>
```

Build Issues using NuGet

To handle a deployment issue related to **SQLite.Interop.dll**, custom .targets file are used in the NuGet package.

If you are having build issues, you can disable this behavior by adding the following property to your project:

```
<DisableSQLiteInteropFix>true</DisableSQLiteInteropFix>
```

Setup and Configuration (Client)

To set up the Reveal Web Client SDK you need to:

1. **Check Dependencies**.

2. **Reference the Web Client SDK**.

3. **Instantiate the Web Client SDK**.

1. Checking Dependencies

The Reveal Web Client SDK has the following 3rd party references:

- jQuery 2.2 or greater
- Day.js 1.8.15 or greater
- Quill RTE 1.3.6 or greater
- Marker Clusterer v3 or greater
- Google Maps v3 or greater

2. Referencing the Web Client SDK

Enabling **$.ig.RevealView** component in a web page requires several scripts to be included. These scripts will be provided as part of Reveal Web Client SDK.

```
<script src="~/Reveal/infragistics.reveal.js"></script>
```

JavaScript files can be found in "<InstallationDirectory>\SDK\Web\JS\Client".

3. Instantiating the Web Client SDK

Reveal's Dashboard presentation is handled natively through the Web Client SDK.

To get started follow these steps:

1. Define a <div /> element with "id" and invoke the **$.ig.RevealView** constructor.

   Note

   > **Hosting Client-Side and Server-Side Parts Separately** If you want to host client-side and server-side parts on different servers, please read here **before** you continue to next step.

2. Call **$.ig.RVDashboard.loadDashboard** providing the *dashboardId* and success and error handlers.

3. In the success handler instantiate the **$.ig.RevealView** component by passing a selector for the DOM element where the dashboard should be rendered into. Finally you should use the retrieved dashboard and set it to the dashboard property of the **$.ig.RevealView**

Sample Code

```
<!DOCTYPE html>
<html>
  <head>
    ⋮
    <script type="text/javascript">
      var dashboardId = "dashboardId";

      $.ig.RVDashboard.loadDashboard(
        dashboardId,
        function (dashboard) {
          var revealView = new $.ig.RevealView("#revealView");
          revealView.dashboard = dashboard;
        },
        function (error) {
          //Process any error that might occur here
        }
      );
    </script>
  </head>
  <body>
    <div id="revealView" style="height:500px;" />
  </body>
</html>
```

Creating Your First App

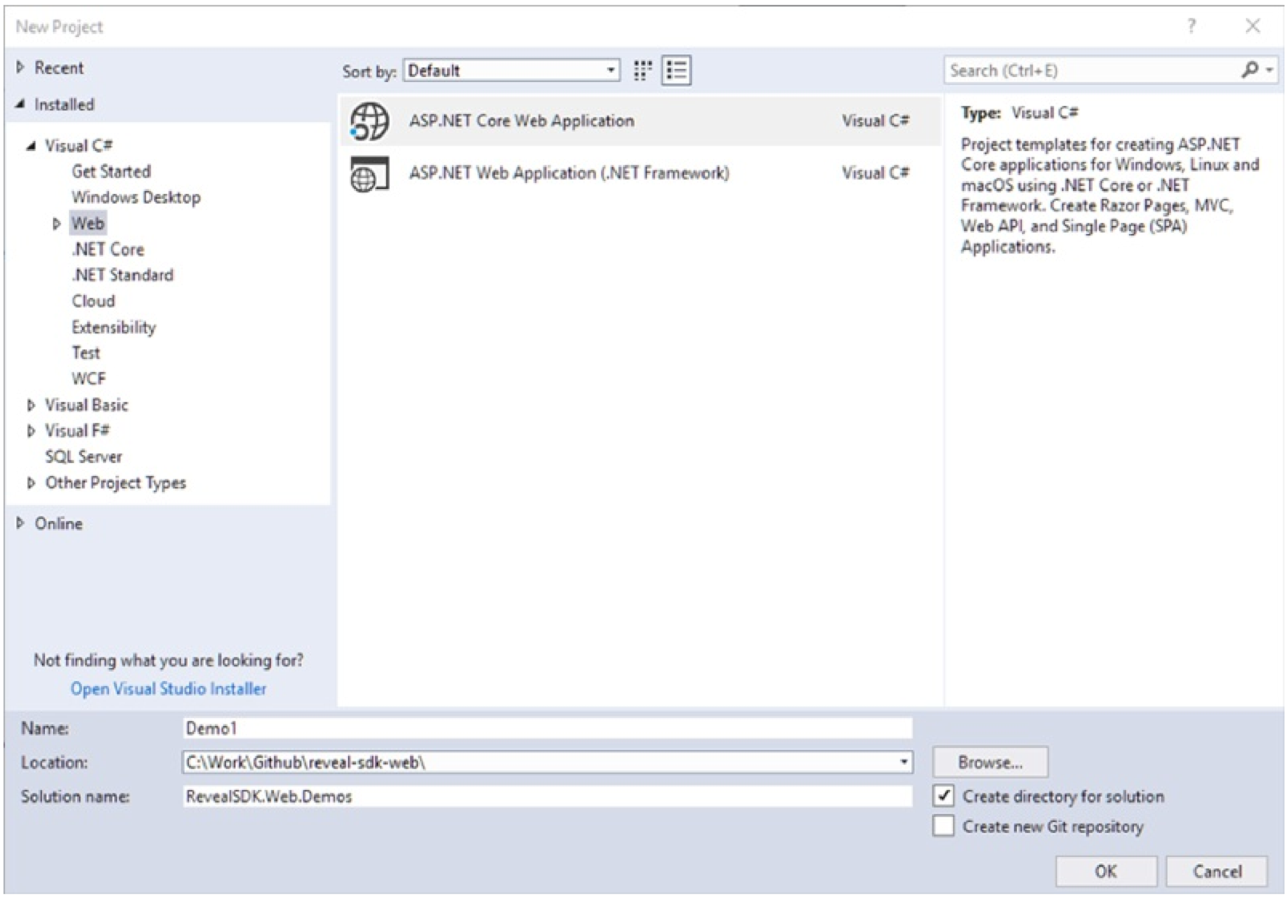
This part aims to guide you through the initial steps of showing a dashboard on your web page/application for the first time.

Steps

1. Creating the Project
2. Installing Reveal SDK
3. Working on Server configuration
4. Embedding Reveal in your Client Application
5. Using Reveal Fonts
6. Styling the Client Application

Step 1 - Create the Project

Open Visual Studio 2017 and create new project of type **ASP.NET Core Web Application**:



Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

Name: Demo1

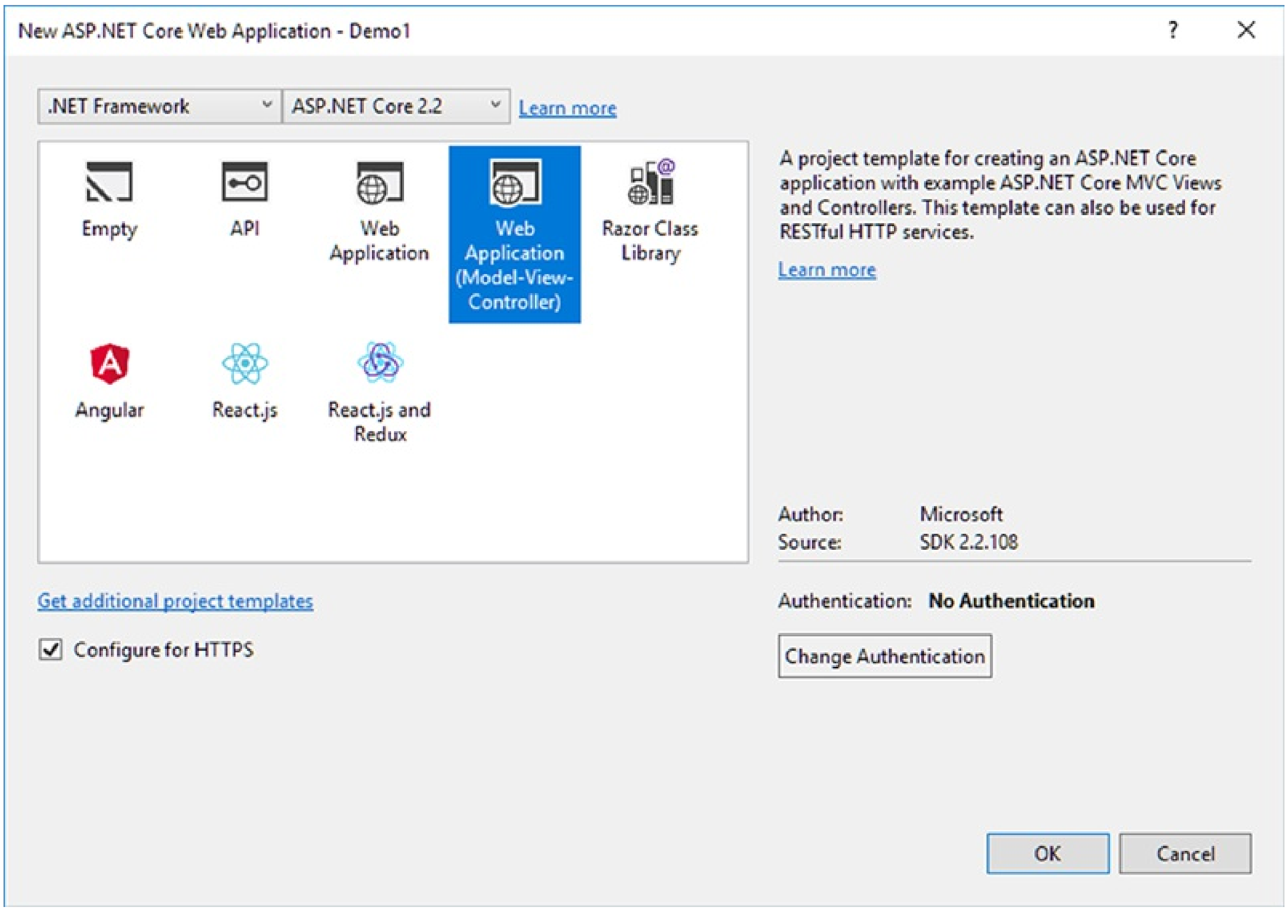
Location: C:\Work\Github\reveal-sdk-web\

Solution name: RevealSDK.Web.Demos

[✓] Create directory for solution

[ ] Create new Git repository

Select **.NET Framework** and **ASP.NET Core 2.2** as follows:


New ASP.NET Core Web Application - Demo1

.NET Framework | ASP.NET Core 2.2 | Learn more

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Learn more

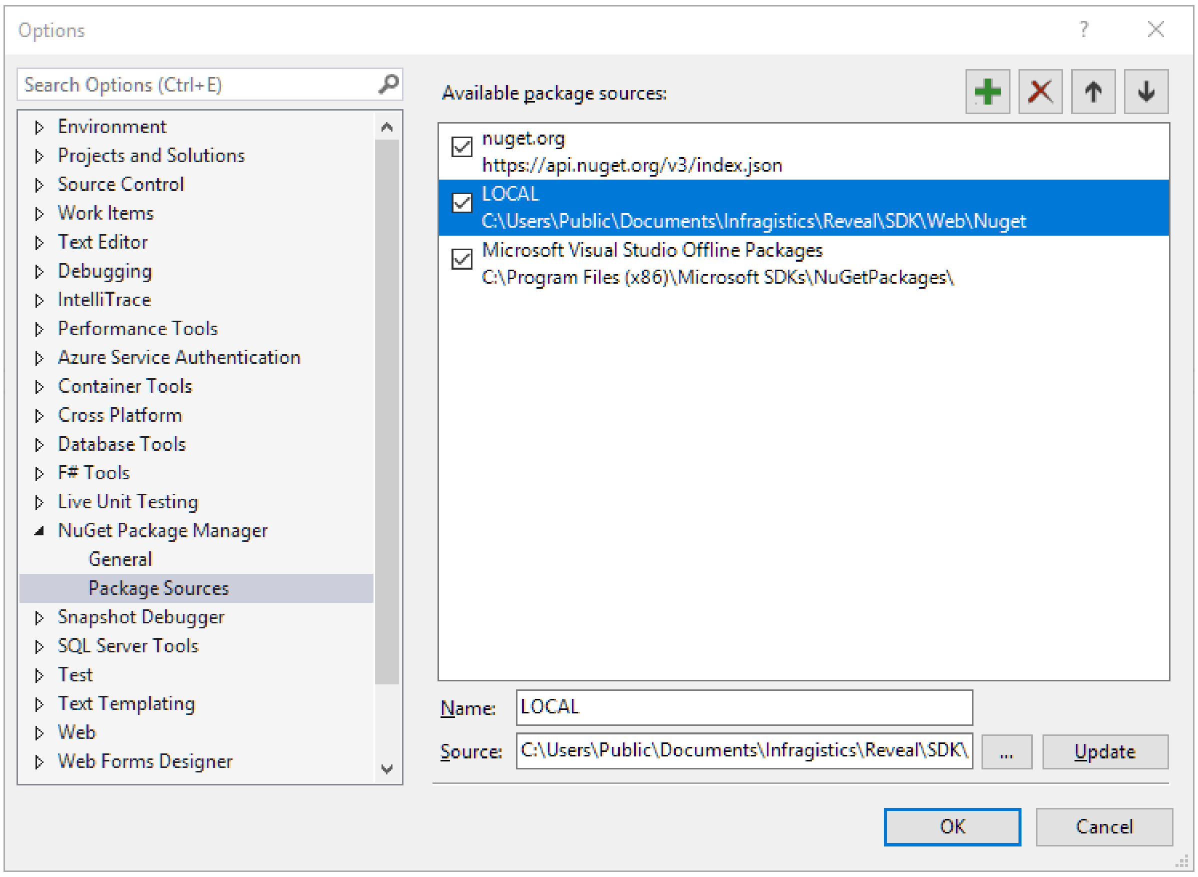
Author: Microsoft
Source: SDK 2.2.108

Authentication: No Authentication

Configure for HTTPS: ✓

Step 2 - Install Reveal SDK

Download the *Infragistics Reveal SDK* from https://www.revealbi.io/ and install it on your machine. In **Visual Studio** go to **Tools > Options > Nuget Package Manager > Package Sources**. Add a new source pointing to the Nuget folder of the installed SDK:


Options

Available package sources:

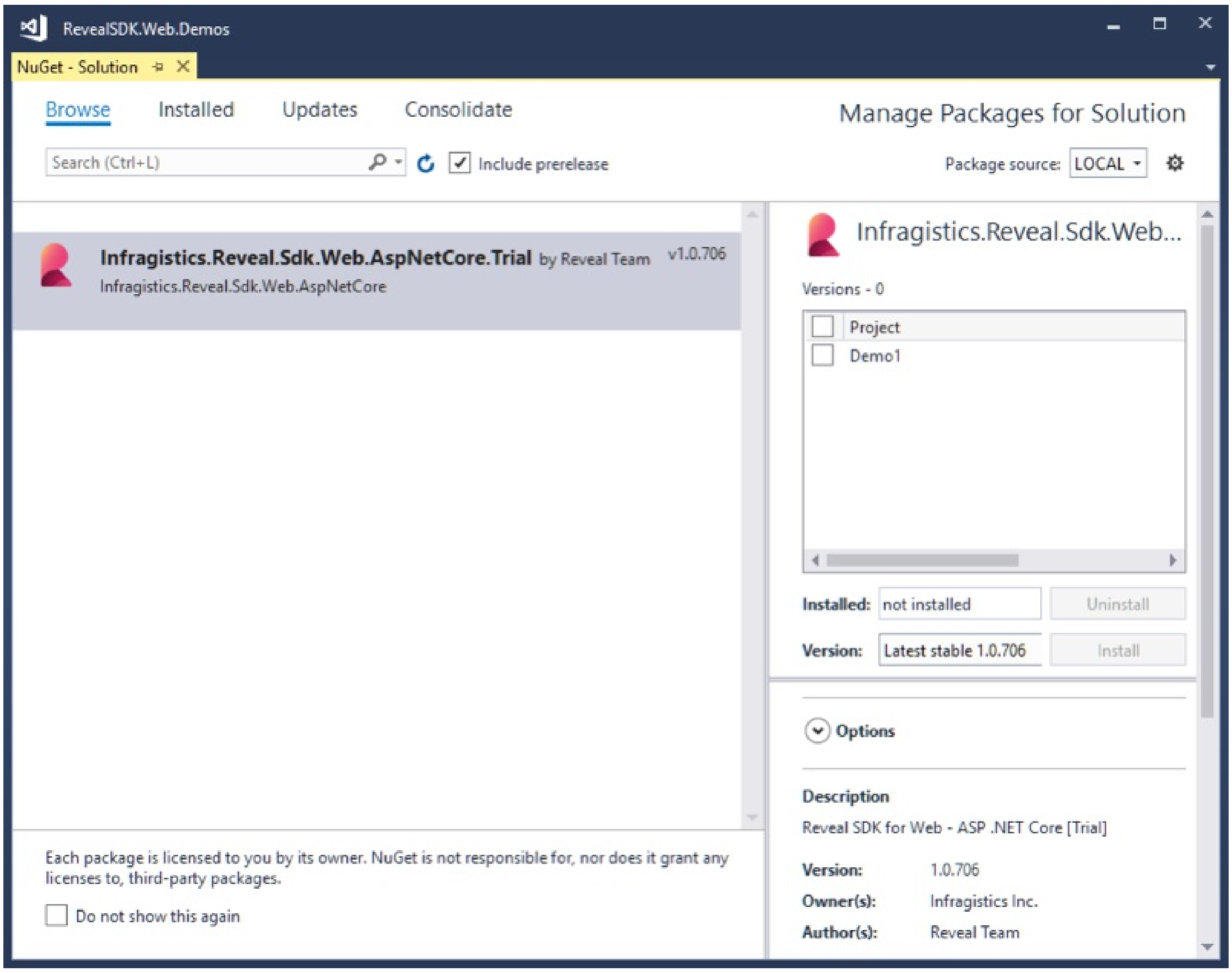
✓ nuget.org
https://api.nuget.org/v3/index.json

✓ LOCAL
C:\Users\Public\Documents\Infragistics\Reveal\SDK\Web\Nuget

✓ Microsoft Visual Studio Offline Packages
C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

Name: LOCAL
Source: C:\Users\Public\Documents\Infragistics\Reveal\SDK\

After that you can install the Nuget by changing the package source to the one you added:



Infragistics.Reveal.Sdk.Web.AspNetCore.Trial by Reveal Team v1.0.706
Infragistics.Reveal.Sdk.Web.AspNetCore

Infragistics.Reveal.Sdk.Web...

Versions - 0

Project
Demo1

Installed: not installed

Version: Latest stable 1.0.706

Description

Reveal SDK for Web - ASP .NET Core [Trial]

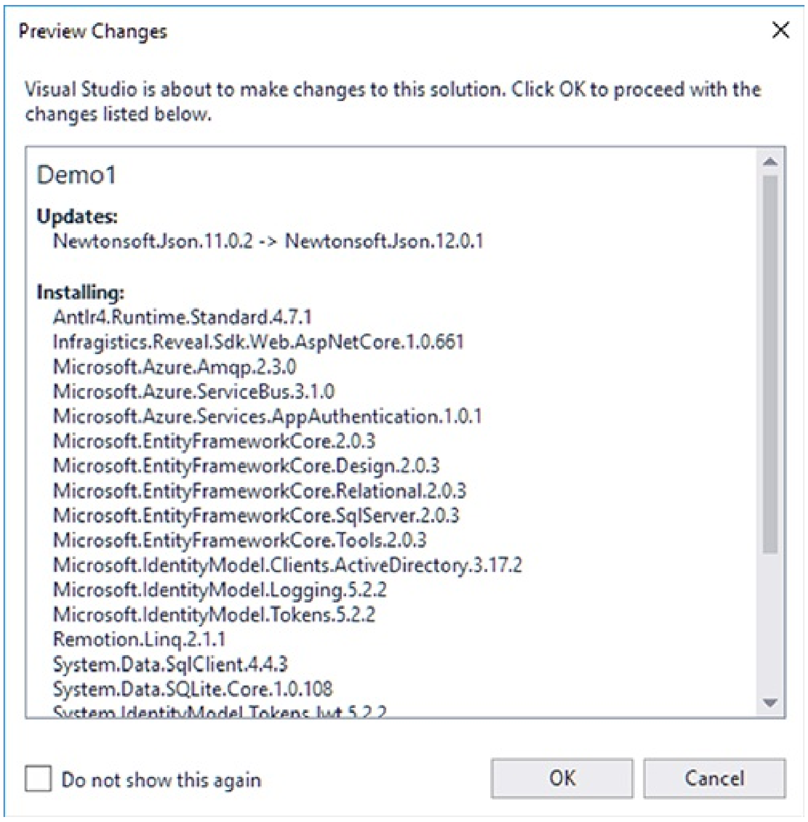
Version: 1.0.706

Owner(s): Infragistics Inc.

Author(s): Reveal Team

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again



Preview Changes

Visual Studio is about to make changes to this solution. Click OK to proceed with the changes listed below.

Demo1

Updates:
Newtonsoft.Json.11.0.2 -> Newtonsoft.Json.12.0.1

Installing:
Antlr4.Runtime.Standard.4.7.1
Infragistics.Reveal.Sdk.Web.AspNetCore.1.0.661
Microsoft.Azure.Amqp.2.3.0
Microsoft.Azure.ServiceBus.3.1.0
Microsoft.Azure.Services.AppAuthentication.1.0.1
Microsoft.EntityFrameworkCore.2.0.3
Microsoft.EntityFrameworkCore.Design.2.0.3
Microsoft.EntityFrameworkCore.Relational.2.0.3
Microsoft.EntityFrameworkCore.SqlServer.2.0.3
Microsoft.EntityFrameworkCore.Tools.2.0.3
Microsoft.IdentityModel.Clients.ActiveDirectory.3.17.2
Microsoft.IdentityModel.Logging.5.2.2
Microsoft.IdentityModel.Tokens.5.2.2
Remotion.Linq.2.1.1
System.Data.SqlClient.4.4.3
System.Data.SQLite.Core.1.0.108
System.IdentityModel.Tokens.Jwt.5.2.2

Do not show this again

Step 3 - Work on Server Configuration

Create a new *Reveal SDK* folder in the project and add the **RevealSdkContext.cs** class, which implements the **RevealSdkContextBase** abstract class:

```
using Reveal.Sdk;
using System;
using System.IO;
using System.Reflection;
using System.Threading.Tasks;

namespace Demo1.RevealSDK
{
    public class RevealSdkContext : RevealSdkContextBase
    {
        public override IRVDataSourceProvider DataSourceProvider => null;

        public override IRVDataProvider DataProvider => null;

        public override IRVAuthenticationProvider AuthenticationProvider => null;

        public override Task<Dashboard> GetDashboardAsync(string dashboardId)
        {
            var dashboardFileName = dashboardId +".rdash";
            var resourceName = $"Demo1.Dashboards.{dashboardFileName}";
            var assembly = Assembly.GetExecutingAssembly();
            return Task.FromResult(new Dashboard(assembly.GetManifestResourceStream(resourceName)));
        }

        public override Task SaveDashboardAsync(string userId, string dashboardId, Dashboard dashboard)
        {
            return Task.CompletedTask;
        }

    }
}
```

In the code above **Demo1.Dashboards** indicates the location where the dashboard files will be contained, so let's create a new Dashboards folder in the project and leave it empty for now.

To do this, add the following code to **ConfigureServices** method in **Startup.cs**:

```
services.AddRevealServices(new RevealEmbedSettings
{
    CachePath = @"C:\Temp"
}, new RevealSdkContext());

services.AddMvc().AddReveal();
```

And the necessary references in the same file:

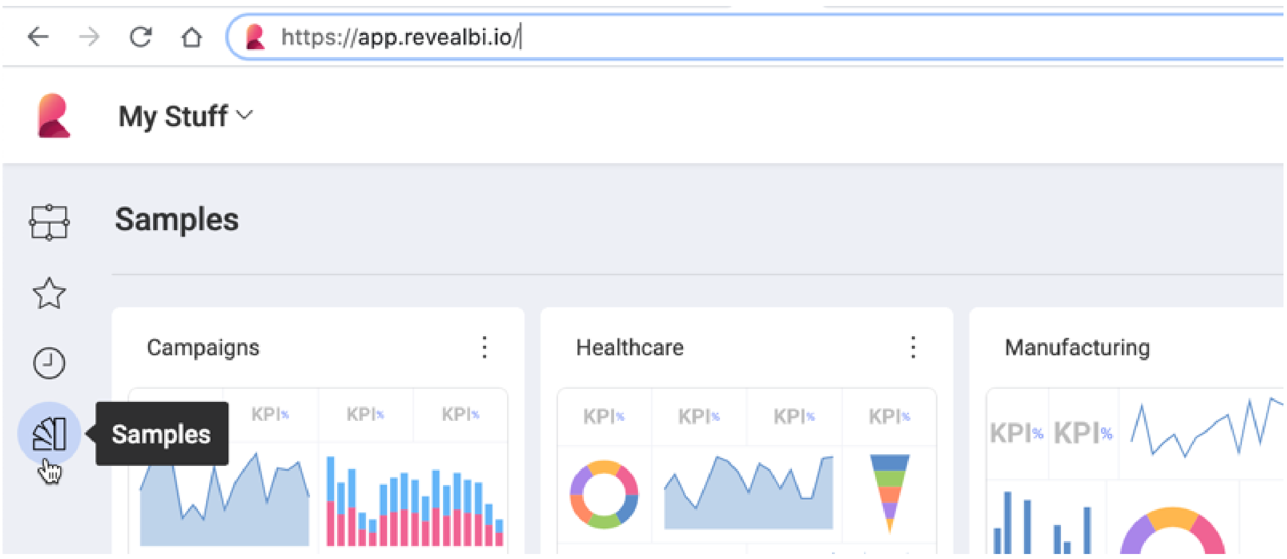
```
using Demo1.RevealSDK;
using Reveal.Sdk;
```

If you experience any issues, please refer to the **Setup and Configuration (Web)** topic.

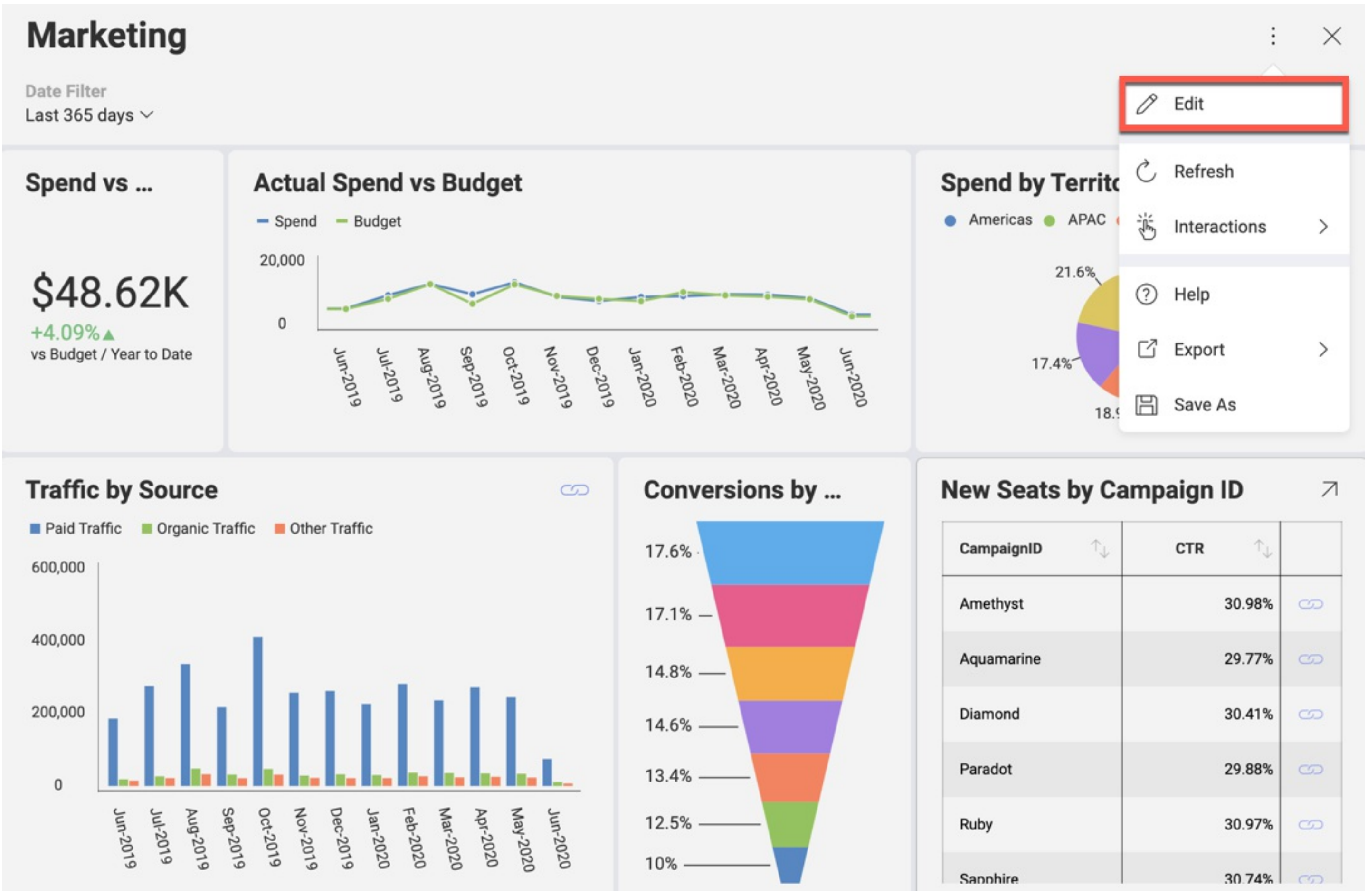
Step 4 - Embed Reveal in your Client application

Let's start this step by getting a dashboard ready. For the purpose of this demo, you can use the **Marketing dashboard** from the **Samples** section in Reveal, but with a different theme.

Open the Reveal app (https://app.revealbi.io) and go to the **Samples**.

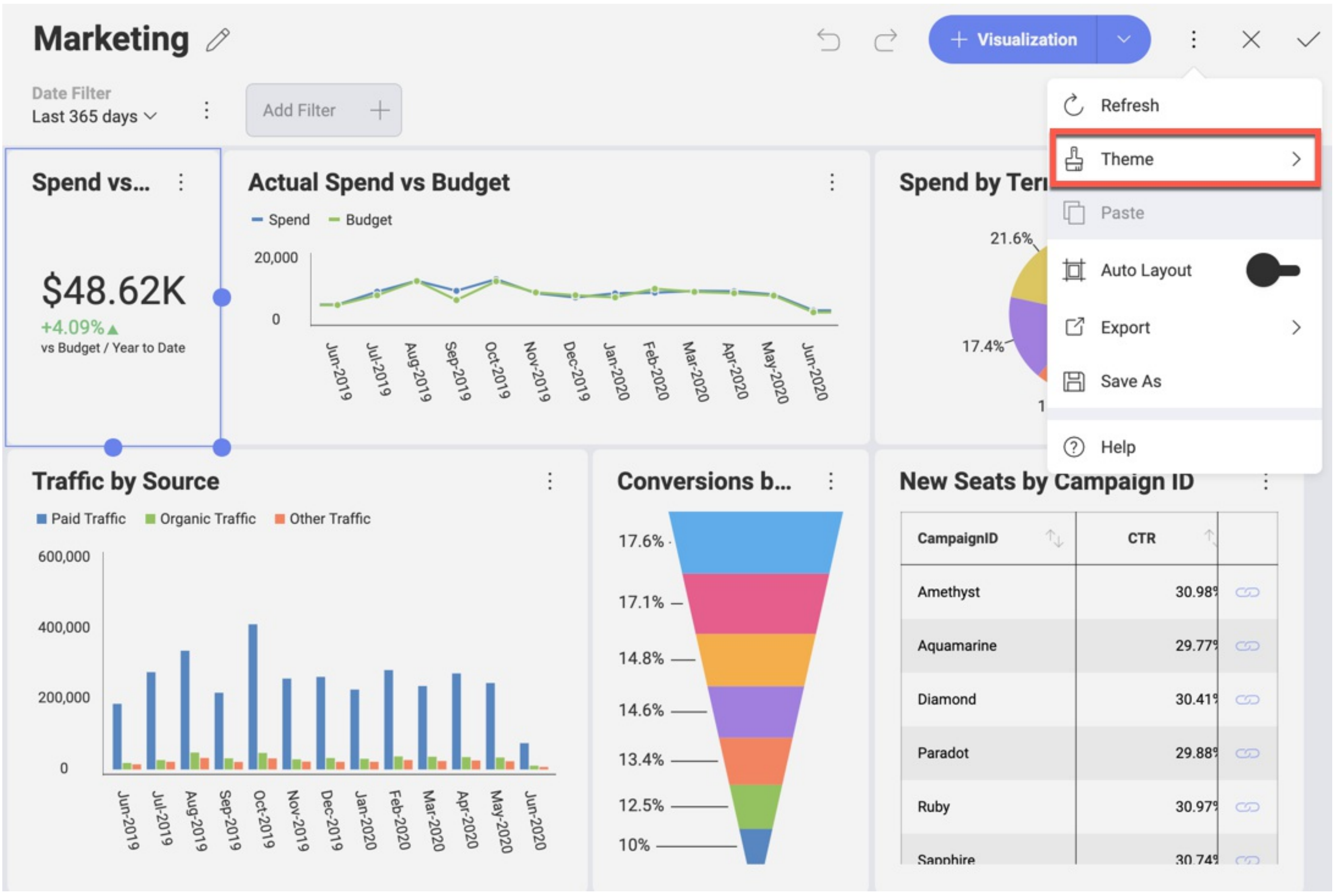


Select the Marketing dashboard and enter **edit mode**:



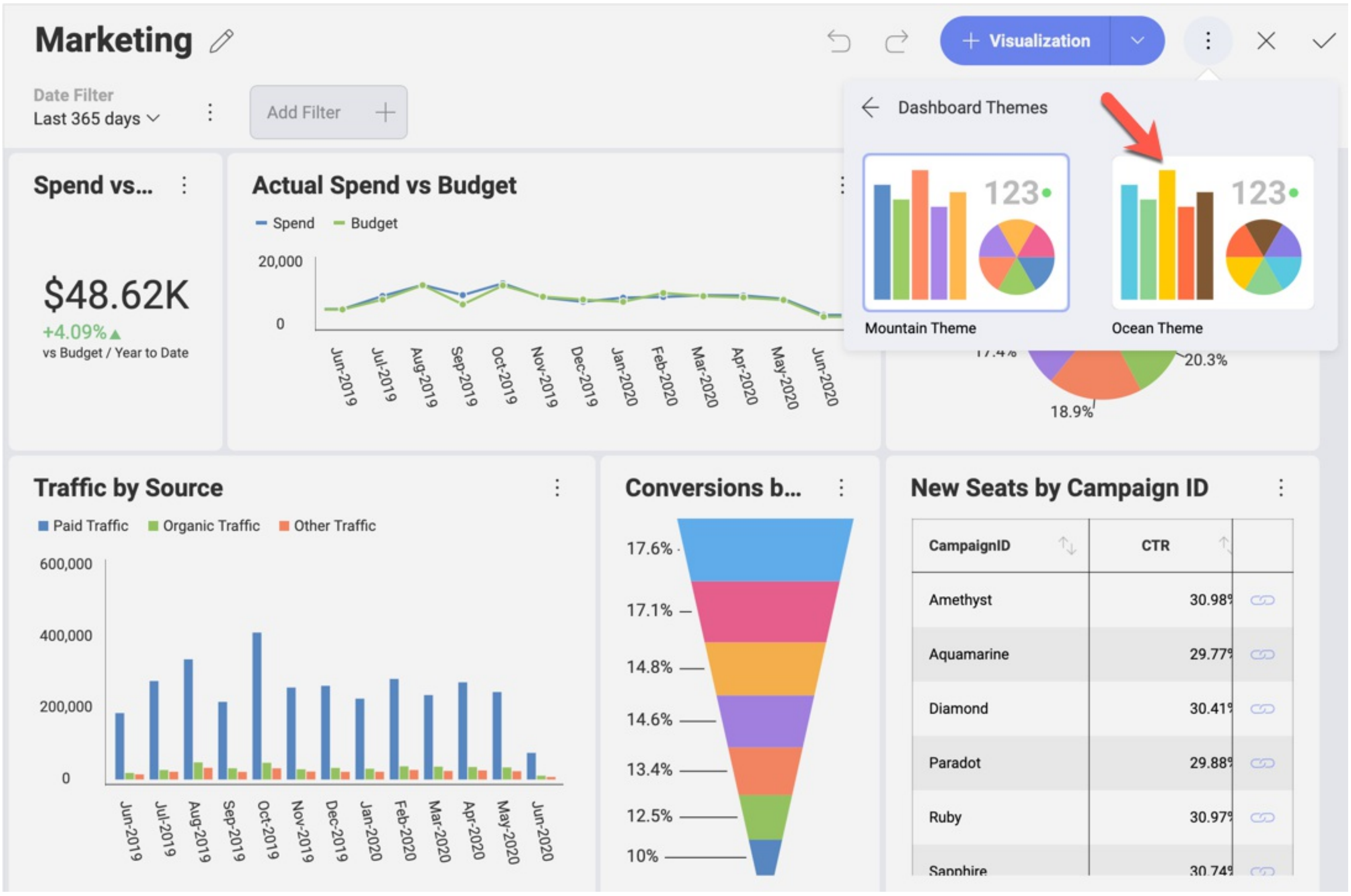
CampaignID	CTR
Amethyst	30.98%
Aquamarine	29.77%
Diamond	30.41%
Paradot	29.88%
Ruby	30.97%
Sapphire	30.74%

Once in Edit mode click the *Theme* button:



CampaignID	CTR
Amethyst	30.98%
Aquamarine	29.77%
Diamond	30.41%
Paradot	29.88%
Ruby	30.97%
Sapphire	30.74%

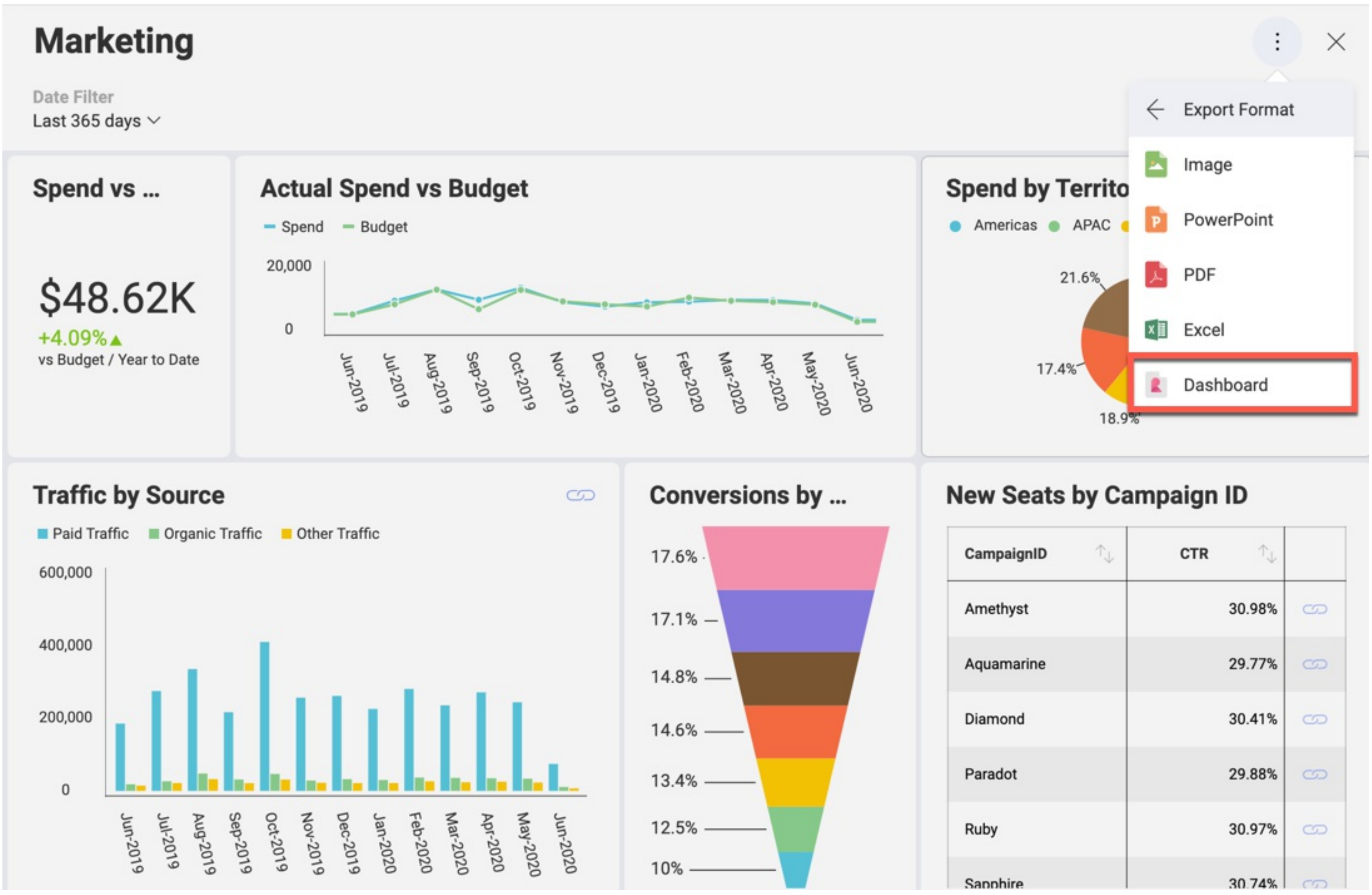
And choose the *Ocean Theme*:



Save the modified dashboard and *Export* it:

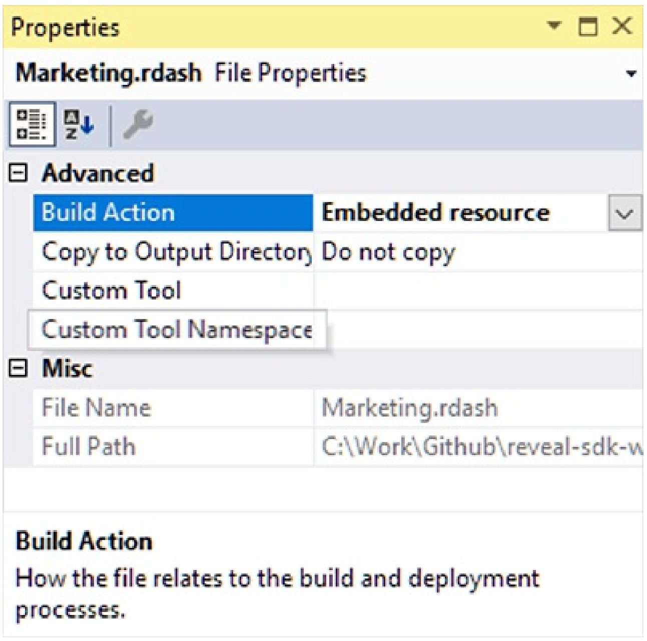
Note

As the Marketing dashboard is part of the Reveal App **Samples**, you cannot save it the same way as a regular dashboard. Instead, you need to use **Save As** and choose a location.



CampaignID	CTR
Amethyst	30.98%
Aquamarine	29.77%
Diamond	30.41%
Paradot	29.88%
Ruby	30.97%
Sapphire	30.74%

Move the **Marketing.rdash** dashboard file to the Dashboards folder, which you created in step 3, and set the Build Action for this item to **Embedded resource** in Visual Studio:



Advanced

Build Action	Embedded resource
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	

Misc

File Name	Marketing.rdash
Full Path	C:\Work\Github\reveal-sdk-w

Build Action
How the file relates to the build and deployment processes.

Now let’s add a new page *Marketing.cshtml* **to the Views folder** in order to visualize the downloaded dashboard:

```
@{
    ViewData["Title"] = "Marketing";
 }

 @section Scripts
     {
     <script type="text/javascript">
         // Load dashboard in #revealView element
     </script>
 }

 <section>
     <div id="revealView" style="height:800px;"></div>
 </section>
```

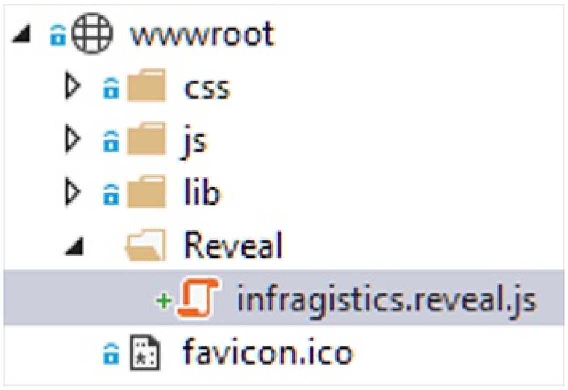
Then, add a new action method in **HomeController.cs**:

```
public IActionResult Marketing()
 {
     return View();
 }
```

Let's add some references to scripts & css files for some third party dependencies of Reveal in **_Layout.cshtml** :

```
<script src="https://unpkg.com/dayjs"></script>
<link rel="stylesheet" href="https://code.jquery.com/ui/1.10.2/themes/smoothness/jquery-ui.css" />
<script type="text/javascript" src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>
<script src="https://cdn.quilljs.com/1.3.6/quill.js"></script>
<link href="https://cdn.quilljs.com/1.3.6/quill.snow.css" rel="stylesheet">
```

To continue, create a new Reveal folder in the *wwwroot* folder of the project. Copy there **infragistics.reveal.js**, which you can find in the **<InstallationDirectory>\SDK\Web\JS\Client** of the *Reveal SDK*:



And then reference this library in **_Layout.cshtml** after the scripts for Day.js:

```
<script src="~/Reveal/infragistics.reveal.js"></script>
```

In the same file, also remove the footer section and add a link in the navigation for the new page:

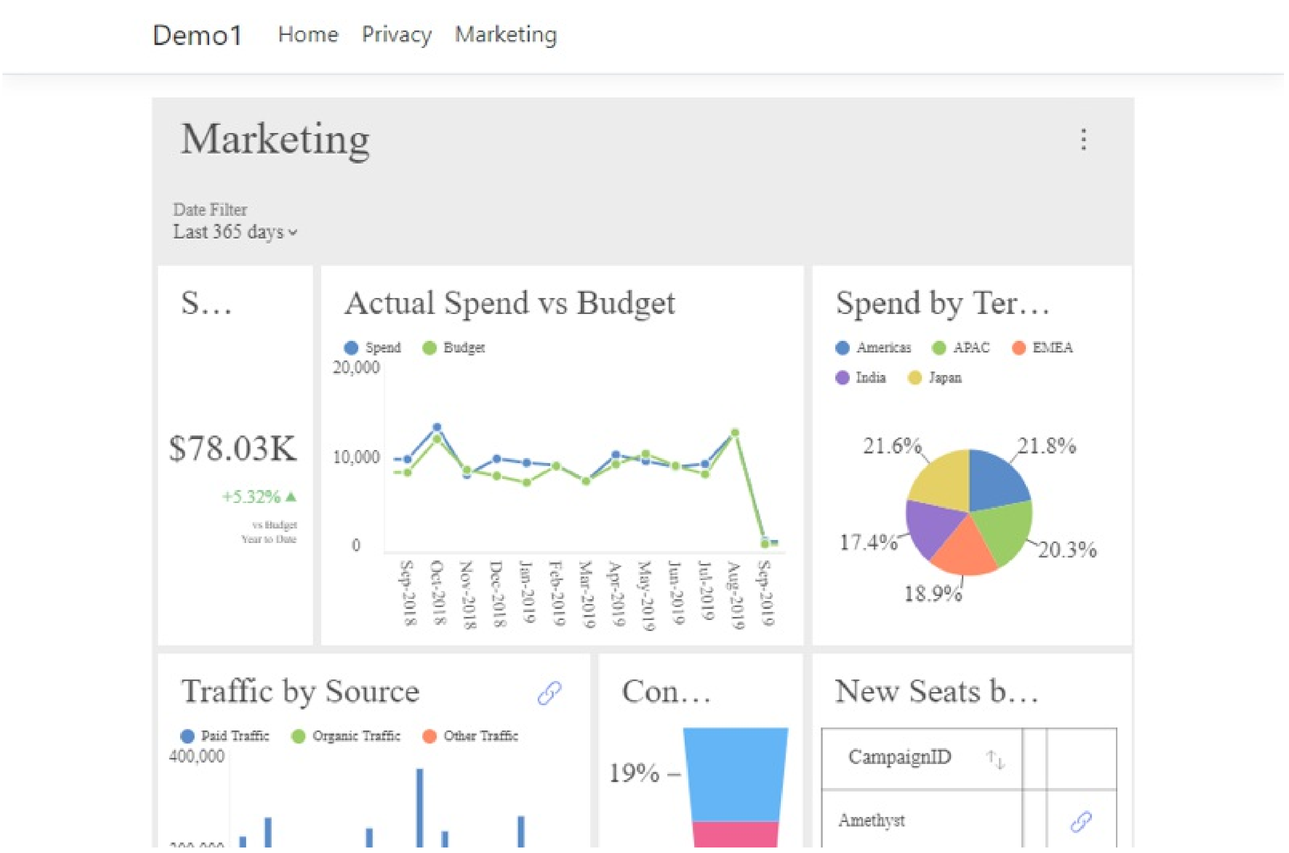
```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Marketing">Marketing</a>
</li>
```

Let's update the script in **Marketing.cshtml** with the logic for loading the dashboard:

```
var dashboardId = "Marketing.rdash";

$.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {
    var revealView = new $.ig.RevealView("#revealView");
    revealView.dashboard = dashboard;
}, function (error) {
    //Process any error that might occur here
});
```

Finally, when running the web page, you can see the dashboard:



If you experience any issues, please refer to the **Setup and Configuration (Web)** topic.

Step 5 - Use Reveal Fonts

Reveal app uses Roboto fonts. In order to achieve the same look as in the app, download the fonts from https://fonts.google.com/specimen/Roboto and copy the following TTF files to the **wwwroot/css** folder of your project:

- Roboto-Regular.ttf
- Roboto-Bold.ttf
- Roboto-Light.ttf
- Roboto-Medium.ttf

Then, add references in the **site.css** as follows:

```
@font-face {
  font-family: "Roboto-Regular";
  src: url("Roboto-Regular.ttf");
}

@font-face {
  font-family: "Roboto-Bold";
  src: url("Roboto-Bold.ttf");
}

@font-face {
  font-family: "Roboto-Light";
  src: url("Roboto-Light.ttf");
}

@font-face {
  font-family: "Roboto-Medium";
  src: url("Roboto-Medium.ttf");
}
```

For font loading improvements add a reference to the Google Web Font Loader in **_Layout.cshtml** next to the infragistics.reveal.js reference:

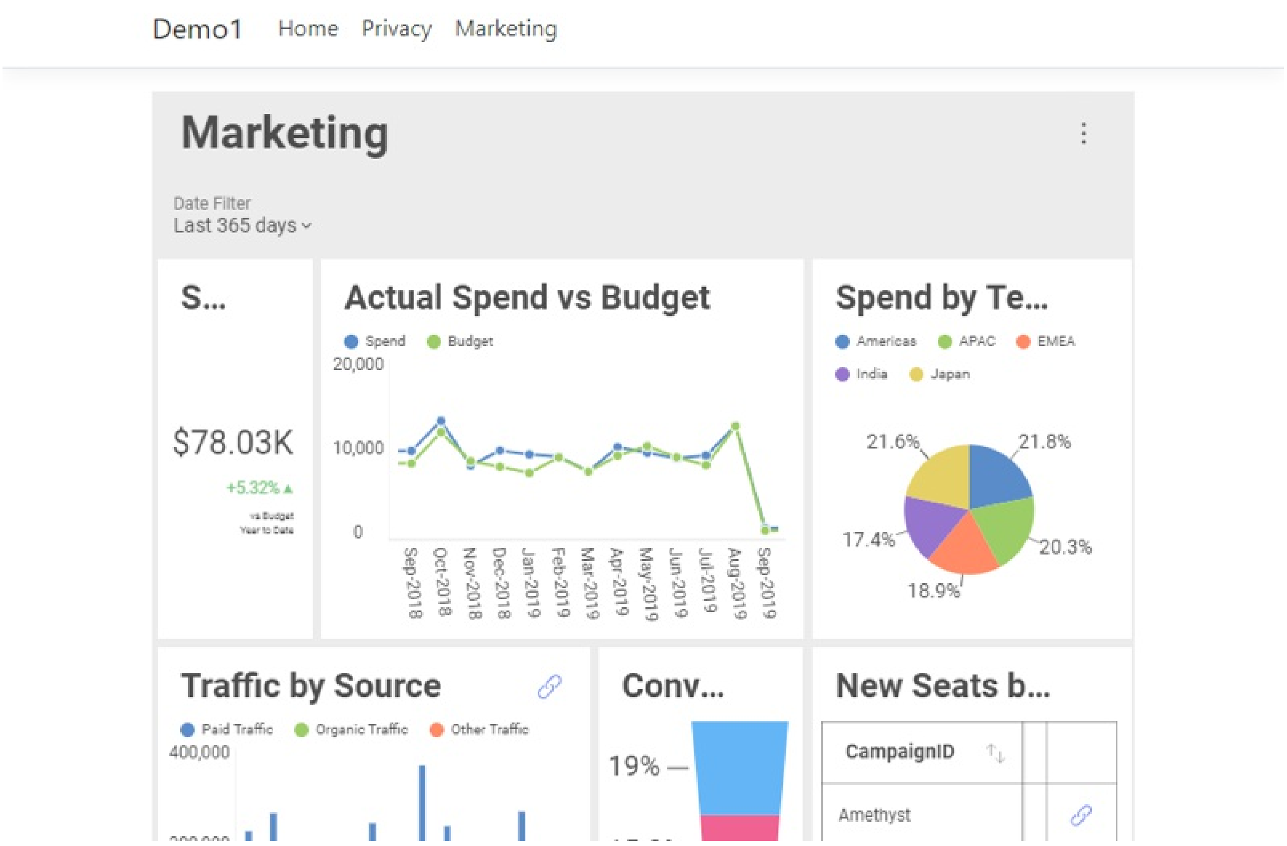
```
<script src="https://ajax.googleapis.com/ajax/libs/webfont/1/webfont.js"></script>
```

Finally, modify the script section of the **Marketing.cshtml** page to take advantage of the font loader:

```
  WebFont.load({
      custom: {
          families: ['Roboto-Regular', 'Roboto-Bold', 'Roboto-Light', 'Roboto-Medium'],
          urls: ['/css/site.css']
      },
      active: function () {
          var dashboardId = "Marketing.rdash";

          $.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {
              var revealView = new $.ig.RevealView("#revealView");
              revealView.dashboard = dashboard;
          }, function (error) {
              //Process any error that might occur here
          });
      },
   });
```

Voila!



Step 6 - Style the Client Application

Instead of using the default template, you can style the Client application.

Remove the Privacy from **HomeController.cs** and modify the Index to redirect to Marketing:

```
public IActionResult Index()
 {
    return RedirectToAction("Marketing");
 }
```

Also, remove the *Index.cshtml* and *Privacy.cshtml* files since they won't be used. Remove the style setting for the <div> element in *Marketing.cshtml*.

Create a new img folder in *wwwroot* and copy there the **logo.png**, which you can download from here.

In **_Layout.cshtml** make the following changes:

- Change the title from *Demo1* to *Overview*
- Remove the div after the header
- Modify the header by adding logo, separator and title:

```
<header>
    <div class="header">
       <img class="logo" src="~/img/logo.png" alt="logo" />
       <span class="line" />
       <h1>Overview</h1>
    </div>
 </header>
```

In **site.css** remove all the styles, except the ones we added for the *Roboto* fonts and add styles for the header:

```
/* Header
 -------------------------------------------------- */

 header {
     display: flex;
     width: 100%;
     height: 70px;
     box-shadow: 0 4px 12px 0 rgba(0, 0, 0, 0.2);
     background-color: #37405a;
 }

 img.logo {
     width: 50px;
     height: 50px;
     margin: 10px;
     float: left;
 }

 span.line {
     float: left;
     width: 1px;
     height: 50px;
     margin-top: 10px;
     border: solid 1px #2b2e40;
 }

 h1 {
     float: left;
     padding-top: 12px;
     padding-left: 20px;
     height: 24px;
     font-family: Roboto-Regular;
     font-size: 20px;
     font-weight: 400;
     color: #ffffff;
 }
```

And styles for the body:

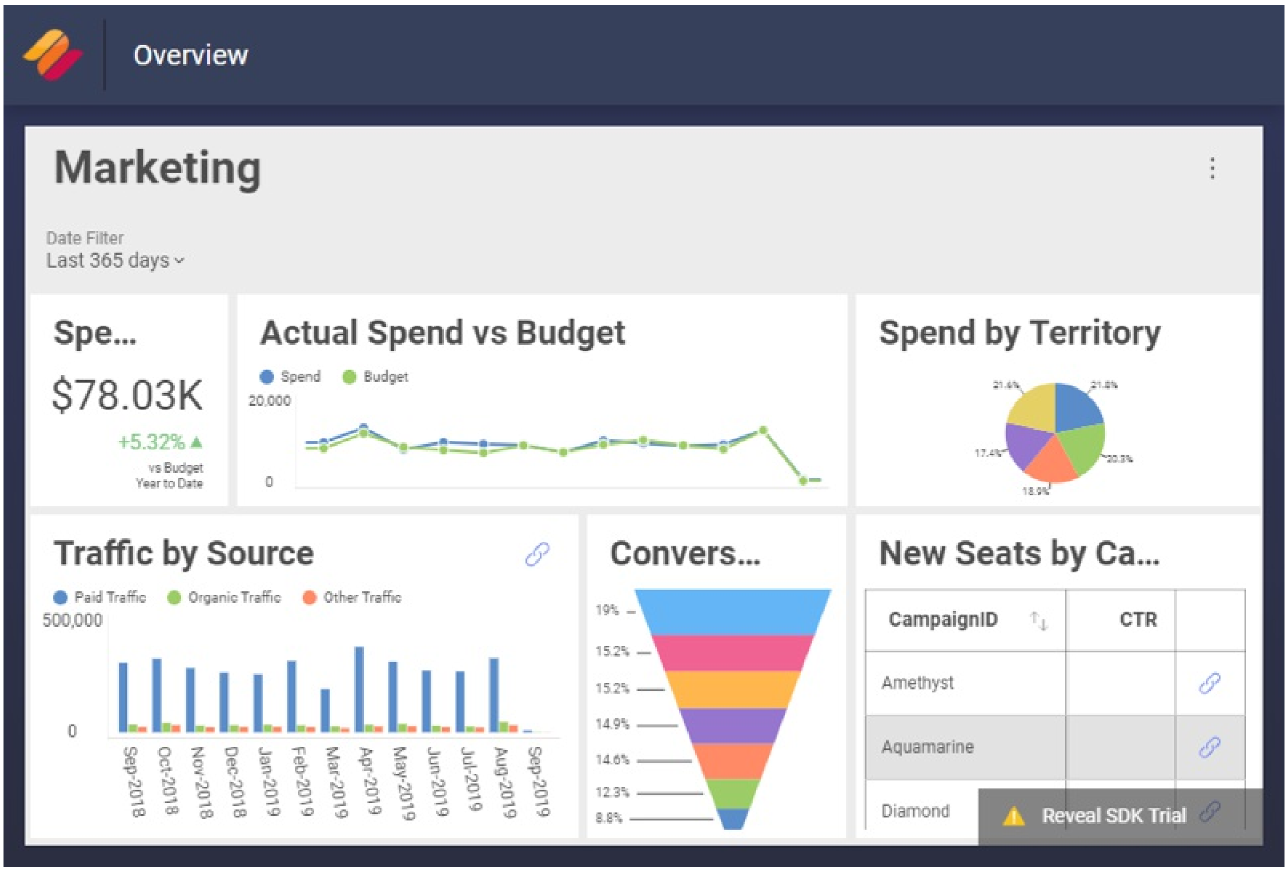
```
/* Body
 -------------------------------------------------- */
 body {
     display: flex;
     flex-direction: column;
     background-image: linear-gradient(to bottom, #30365a, #2b2e40);
 }

 html, body {
     width: 100%;
     height: 100%;
 }

     body section {
         display: block;
         width: 100%;
         height: 100%;
         padding: 15px;
     }

 #revealView {
     height: 100%;
 }
```

And this should be your result:



CampaignID	CTR	
Amethyst		
Aquamarine		
Diamond		

Loading Dashboard Files

Overview

There are two ways to open/save dashboards with the SDK:

- **Server-side**: First, you specify a dashboard ID in the client page. Second, on the server, using a callback method detailed below, you return the stream with the contents of the dashboard with the specified ID.

  Please note that this is the easiest approach and the one recommended when you are first evaluating the SDK.

- **Client-side**: Here you have full control and more flexibility. You provide the stream with the contents of the dashboard on the client page, getting the contents from your own server.

  Using this approach you can, for example, check user permissions, display your own user interface to select the dashboard, or allow users to upload the ".rdash" file to use. For further details about the client-side approach, follow this **Setup and Configuration(Client)**.

The Server-Side Approach

In order to visualize a dashboard, you can provide the SDK with an instance of a Dashboard class, which you could instantiate passing a stream to a rdash or json string representation of a rdash.

The code snippet below shows how to load a .rdash file that is added to the project as an embedded resource. Please note that this method is the implementation for **RevealSdkContextBase.GetDashboardAsync**.

Code

```
public override Task<Dashboard> GetDashboardAsync(string dashboardId)
{
    var dashboardFileName = dashboardId + ".rdash";
    var resourceName = $"Demo1.Dashboards.{dashboardFileName}";
    var assembly = Assembly.GetExecutingAssembly();
    var rdashStream = assembly.GetManifestResourceStream(resourceName)
    var dashboard = new Dashboard(rdashStream);

    return Task.FromResult(dashboard);
}
```

This code for **RevealSdkContextBase.GetDashboardAsync** will be invoked on the server when you use **RVDashboard.loadDashboard** function on the client. And you will get the *dashboardId* that was specified client-side as the first parameter.

Replacing Data Sources (MS SQL Server)

Overview

Before loading and processing the data for a dashboard (by Reveal Server SDK), you can override the configuration or data to be used for each visualization of the dashboard.

One of the properties to implement in **RevealSdkContextBase** is:

```
IRVDataSourceProvider DataSourceProvider { get; }
```

A class implementing the interface **IRVDataSourceProvider** may replace or modify the data source used by a given visualization or dashboard filter.

Use Cases

Below you can find a list of common use cases:

- You can change the name of the database being used, depending on the current user, or any other attributes your app might get like userId, division, company, customer, etc. By doing this, you can have a single dashboard getting data from a multi-tenant database.
- You can change the name of the table being used, the path of the file to load, etc. The use case is similar to the one described above.
- You can replace a data source with an in-memory data source. As the Reveal App doesn't support in-memory data sources, you can design a dashboard using a CSV file and then use this callback to replace the CSV data source with an in-memory one. In this scenario, data is actually loaded from memory (or with your custom data loader). For further details about how to use in-memory data sources, refer to **In-Memory Data Support**.

Code

The following code snippet shows an example of how to replace the data source for visualizations in the dashboard. The method **ChangeVisualizationDataSourceItemAsync** will be invoked for every visualization, on every single dashboard being opened.

```
public class SampleDataSourceProvider : IRVDataSourceProvider
{
    public Task<RVDataSourceItem> ChangeDashboardFilterDataSourceItemAsync(string userId, string dashboardId, RVDashboardFilter globalFilter,
RVDataSourceItem dataSourceItem)
    {
        return Task.FromResult<RVDataSourceItem>(null);
    }

    public Task<RVDataSourceItem> ChangeVisualizationDataSourceItemAsync(
        string userId, string dashboardId, RVVisualization visualization,
        RVDataSourceItem dataSourceItem)
    {
        var sqlServerDsi = dataSourceItem as RVSqlServerDataSourceItem;
        if (sqlServerDsi != null)
        {
            // Change SQL Server host and database
            var sqlServerDS = (RVSqlServerDataSource)sqlServerDsi.DataSource;
            sqlServerDS.Host = "10.0.0.20";
            sqlServerDS.Database = "Adventure Works";

            // Change SQL Server table/view
            sqlServerDsi.Table = "Employees";
            return Task.FromResult((RVDataSourceItem)sqlServerDsi);
        }

        // Fully replace a data source item with a new one
        if (visualization.Title == "Top Customers")
        {
            var sqlDs = new RVSqlServerDataSource();
            sqlDs.Host = "salesdb.local";
            sqlDs.Database = "Sales";

            var sqlDsi = new RVSqlServerDataSourceItem(sqlDs);
            sqlServerDsi.Table = "Customers";

            return Task.FromResult((RVDataSourceItem)sqlServerDsi);
        }

        return Task.FromResult(dataSourceItem);
    }
}
```

In the example above, the following two replacements will be performed:

- All data sources using a MS SQL Server database will be changed to use the hardcoded server “10.0.0.20”, the “Adventure Works” database, and the “Employees” table.

  Note: This is a simplified scenario, replacing all visualizations to get data from the same table, which makes no sense as a real world scenario. In real-world applications, you’re probably going to use additional information like userId, dashboardId, or the values in the data source itself (server, database, etc.) to infer the new values to be used.

- All widgets with the title “Top Customers” will have their data source set to a new SQL Server data source, getting data from the “Sales” database in the “salesdb.local” server, using the table “Customers”.

Please note that in addition to implement **IRVDataSourceProvider** you need to modify your implementation of **RevealSdkContextBase.DataSourceProvider** to return it:

```
IRVDataSourceProvider DataSourceProvider => new SampleDataSourceProvider();
```

Replacing Excel and CSV file DataSources

Overview

When we create a dashboard in **Reveal Application** we often use Excel or CSV files stored in the cloud to populate it with data.
After exporting the dashboard and embedding it in custom application, we can move these files in a local directory and then use the **Reveal SDK** to access and set them as a local datasource.

Steps

To populate the exported dashboard using local Excel and CSV files, you need to follow these steps:

1. **Export the dashboard** file as explained in **Getting Dashboards for the SDK**
2. **Load the dashboard** in your application as described in **Creating Your First App**
3. **Download the files** you used to create the dashboard from your cloud storage and copy them to a local folder.
4. **Set the local folder name** as a value of the *LocalStoragePath*. Details about this you can find here: **Setup and Configuration(Server) - Initializing the Server SDK**
5. **Add a new *CloudToLocalDatasourceProvider* class** in the project.
6. **Copy the implementation code** from the relevant snippet in **Code** section below.
7. **Set the *DataSourceProvider* property** of the *RevealSdkContext* class to *CloudToLocalDatasourceProvider*:

```
public override IRVDataSourceProvider DataSourceProvider => new CloudToLocalDatasourceProvider();
```

Code

```
public class CloudToLocalDatasourceProvider : IRVDataSourceProvider
{
    public Task<RVDataSourceItem> ChangeDashboardFilterDataSourceItemAsync(string userId, string dashboardId,
                RVDashboardFilter filter, RVDataSourceItem dataSourceItem)
    {
        return ProcessDataSourceItem(dataSourceItem);
    }

    public Task<RVDataSourceItem> ChangeVisualizationDataSourceItemAsync(string userId, string dashboardId,
                RVVisualization visualization, RVDataSourceItem dataSourceItem)
    {
        return ProcessDataSourceItem(dataSourceItem);
    }

    protected Task<RVDataSourceItem> ProcessDataSourceItem(RVDataSourceItem dataSourceItem)
    {
        // Return data source unless it is an excel or csv file.
        if (dataSourceItem is RVExcelDataSourceItem == false &&
            dataSourceItem is RVCsvDataSourceItem == false)
        {
            return Task.FromResult(dataSourceItem);
        }

        var resourceBased = dataSourceItem as RVResourceBasedDataSourceItem;
        var resourceItem = resourceBased?.ResourceItem as RVDataSourceItem;
        var title = resourceItem?.Title;

        if (string.IsNullOrEmpty(title))
        {
            return Task.FromResult(dataSourceItem);
        }

        var localItem = new RVLocalFileDataSourceItem();

        // The SDK uses the local folder set as LocalStoragePath.
        localItem.Uri = @"local:/" + title;

        // The title assigned here is the original data source name.
        localItem.Title = title;
        resourceBased.ResourceItem = localItem;

        return Task.FromResult(dataSourceItem);
    }
}
```

Note

The *CloudToLocalDatasourceProvider* replaces automatically Excel and CSV files only. Other file types or data sources will remain unchanged. The replacement files should be the same ones used to create the dashboard or, alternatively, new files that share the **same schema** but with different data.

In-Memory Data Support

Overview

In some cases you need to use data already in memory as part of your application state. Using, for example, the result of a report requested by the user, or information from a data source not yet supported by Reveal (like a custom database or a specific file format).

In-Memory is a special type of data source that can be used only with the SDK and not in the Reveal application out of the box. Because of this, you cannot use an "in-memory data source" directly, you need to take a different approach as explained below.

Using In-Memory Data Source

The recommended approach is to **define a data file with a schema, matching your in-memory data**. Data files can be, for example, CSV or Excel files, and a schema is basically a list of fields and the data type for each field.

In the example below you find details about how to create a data file with a given schema, and then use data in memory instead of getting information from a database.

Code Example

In the following example, you want to use in-memory data with the list of Employees in the company, in order to embed a Reveal dashboard showing HR metrics in your HR system. And instead of getting the list of employees from your database, you want to use data in memory.

To achieve all that, you will need to create and export a dashboard in the Reveal application using dummy data.

About the Reveal Application

The Reveal Application is a self-service business intelligence tool that enables you to create, view and share dashboards with your teams. For further details about the Reveal app, you can access an **online demo** or browse the **Help Documentation**.

Getting the Data File and Sample Dashboard Ready

As simplified Employee has only the following properties:

- *EmployeeID*: string
- *Fullname*: string
- *Wage*: numeric

Steps:

1. Create The CSV file with the same schema:

   ```
   EmployeeID,Fullname,Wage
   23,John Smith,345.67
   45,Emma Thompson,432.23
   ```

2. Upload the file to your preferred File Sharing System, like Dropbox or Google Drive.

3. Create a dashboard within the Reveal app using the dummy data. Please note that you are going to provide the real production data later in your application.

4. Export the dashboard from the Reveal app (Dashboard Menu → Export → Dashboard) and save a .rdash file.

Visualizing the Dashboard and Returning the Actual Data

Now you need to visualize the dashboard using your own data instead of the dummy one.

1. Implement **IRVDataSourceProvider** and return it as the **DataSourceProvider** property in **RevealSdkContextBase**, as described in **Replacing Data Sources**.

   Then, in the implementation for the method **ChangeVisualizationDataSourceItemAsync**, you need to add a code similar to this one:

   ```
   public Task<RVDataSourceItem> ChangeVisualizationDataSourceItemAsync(string userId, string dashboardId, RVVisualization visualization,
   RVDataSourceItem dataSourceItem)
   {
       var csvDsi = dataSourceItem as RVCsvDataSourceItem;
       if (csvDsi != null)
       {
           var inMemDsi = new RVInMemoryDataSourceItem("employees");
           return Task.FromResult((RVDataSourceItem)inMemDsi);
       }
       return Task.FromResult((RVDataSourceItem)null);
   }
   ```

   This way you basically replace all references to CSV files in the dashboard with the in-memory data source identified by “employees”. This identification will be used later when returning the data.

2. Implement the method that will return the actual data, to do that implement **IRVDataProvider** as shown below:

   ```
   public class EmbedDataProvider : IRVDataProvider
   {
       public Task<IRVInMemoryData> GetData(string userId, RVInMemoryDataSourceItem dataSourceItem)
       {
           var datasetId = dataSourceItem.DatasetId;
           if (datasetId == "employees")
           {
               var data = new List<Employee>()
                   {
                       new Employee(){ EmployeeID = "1", Fullname="John Doe", Wage = 80325.61 },
                       new Employee(){ EmployeeID = "2", Fullname="Doe John", Wage = 10325.61 },
                   };
               return Task.FromResult<IRVInMemoryData>(new RVInMemoryData<Employee>(data));
           }
           else
           {
               throw new Exception("Invalid data requested");
           }
       }
   }
   ```

   Please note that the properties in the Employee class are named exactly as the columns in the CSV file, and the data type is also the same. In case you want to alter the field name, field label and/or data type of any of the properties you can use attributes in the class declaration:

   - RVSchemaColumn attribute can be used to alter the field name and/or data type.
   - DisplayName attribute can be used to alter the field label.

```
public class Employee
{
   [RVSchemaColumn("EmployeeID", RVSchemaColumnType.Number)]
   public string EmployeeID { get; set; }

   [DisplayName("EmployeeFullname")]
   public string Fullname { get; set; }

   [RVSchemaColumn("MonthlyWage")]
   public double Wage { get; set; }
}
```

In addition, to implement **IRVDataProvider** you need to modify your implementation of **RevealSdkContextBase.DataProvider** to return it:

```
IRVDataProvider DataProvider => new EmbedDataProvider();
```

Providing Credentials to Data Sources

Overview

The Server SDK allows you to pass in a set of credentials to be used when accessing the data source.

Code

The first step is to implement **IRVAuthenticationProvider** and return it as the **AuthenticationProvider** property in **RevealSdkContextBase**, as shown below.

```
public class EmbedAuthenticationProvider : IRVAuthenticationProvider
{
    public Task<IRVDataSourceCredential> ResolveCredentialsAsync(string userId, RVDashboardDataSource dataSource)
    {
        IRVDataSourceCredential userCredential = null;
        if (dataSource is RVPostgresDataSource)
        {
            userCredential = new RVUsernamePasswordDataSourceCredential("postgresuser", "password");
        }
        else if (dataSource is RVSqlServerDataSource)
        {
            // The "domain" parameter is not always needed and this depends on your SQL Server configuration.
            userCredential = new RVUsernamePasswordDataSourceCredential("sqlserveruser", "password", "domain");
        }
        else if (dataSource is RVGoogleDriveDataSource)
        {
            userCredential = new RVBearerTokenDataSourceCredential("fhJhbUci0mJSUzi1nIiSint....", "user@company.com");
        }
        else if (dataSource is RVRESTDataSource)
        {
            userCredential = new RVUsernamePasswordDataSourceCredential(); // Anonymous
        }
        return Task.FromResult<IRVDataSourceCredential>(userCredential);
    }
}
```

Choosing Which Class to Implement

There are two classes that can be used, both implementing the **IRVDataSourceCredential** interface. You need to choose the class depending on your data source, as detailed below.

- Class **RVBearerTokenDataSourceCredential** works with:
    - Analytics tools (Google Analytics).
    - Content Managers and Cloud Services (Box, Dropbox, Google Drive, OneDrive and SharePoint Online).
- Class **RVUsernamePasswordDataSourceCredential** works with:
    - Customer Relationship Managers (Microsoft Dynamics CRM On-Premises and Online)
    - Databases (Microsoft SQL Server, Microsoft Analysis Services Server, MySQL, PostgreSQL, Oracle, Sybase)
- **Both classes** work with:
    - Other Data Sources (OData Feed, Web Resources, REST API).

No Authentication

Sometimes you might work with an anonymous resource, without authentication. In this particular case, you can use **RVUsernamePasswordDataSourceCredential**, which has an empty constructor. You can do this for any data source that works with the class.

Code snippet to be used with the sample above:

```
else if (dataSource is RVRESTDataSource)
{
    userCredential = new RVUsernamePasswordDataSourceCredential();
}
```

Creating New Visualizations and Dashboards

Overview

As described in **Editing & Saving Dashboards**, there are two ways to handle how you save changes to dashboards: **client-side and server-side**. Those scenarios work fine when users make minor changes to existing dashboards like:

- Adding/modifying filters
- Changing the type of visualization (chart, gauge, grid, etc.)
- Changing the theme

However, to add new visualizations the user needs to **select the data source** to be used. To do that, the containing application needs to provide information to the SDK, so it can display the list of data sources available for a new visualization.

Displaying a List of Data Sources

The callback you need to use to display a list of data sources is **onDataSourcesRequested**. In the case that you don't set your own function to this callback, when a new visualization is created, Reveal will display all data sources used in the dashboard (if any).

Code:

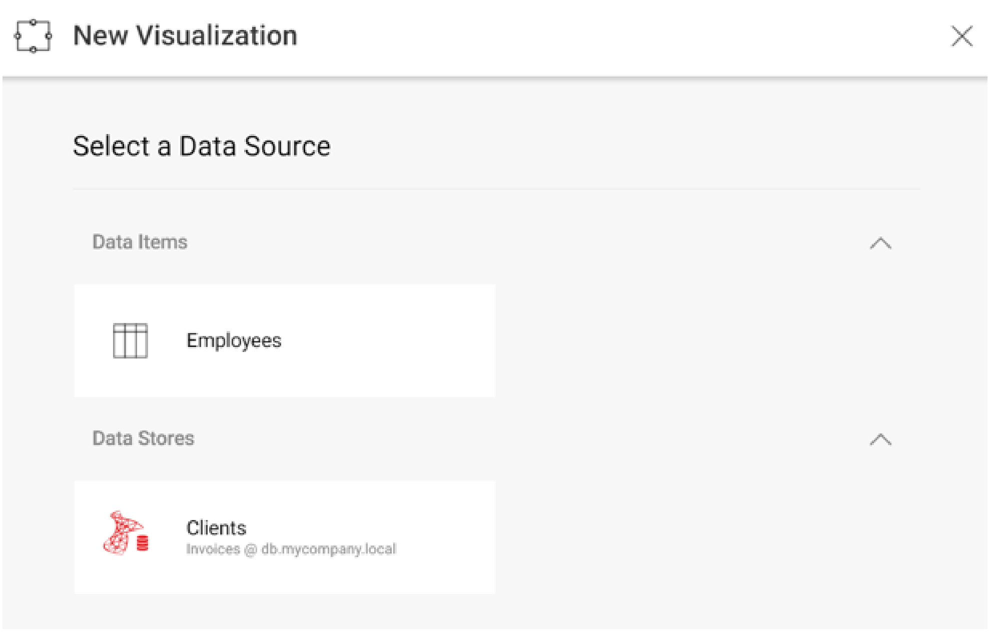
The code below shows how to configure the *data source selection* screen to show an "in-memory" item and a SQL Server data source.

```
window.revealView.onDataSourcesRequested = function (callback) {
    var inMemoryDSI = new $.ig.RVInMemoryDataSourceItem("employees");
    inMemoryDSI.title = "Employees";
    inMemoryDSI.description = "Employees";

    var sqlDs = new $.ig.RVSqlServerDataSource();
    sqlDs.title = "Clients";
    sqlDs.id = "SqlDataSource1";
    sqlDs.host = "db.mycompany.local";
    sqlDs.port = 1433;
    sqlDs.database ="Invoices";

    callback(new $.ig.RevealDataSources([sqlDs], [inMemoryDSI], false));
};
```

The "false" value in the third parameter prevents existing data sources on the dashboard from being displayed. So, when creating a new widget using the "+" button, you should get the following screen:



Please note that the “employees” parameter passed to the “RVInMemoryDataSourceItem” constructor, is the same dataset id used in **In-Memory Data Support** and identifies the dataset to be returned on the server side.

Creating New Dashboards

Creating dashboards from scratch is really simple, you just need to:

- Initialize **$.ig.RevealView** object, without setting the dashboard property to $.ig.RevealView and without using **$.ig.RVDashboard.loadDashboard**;
- Set *startInEditMode* to true, to start the dashboard in edit mode:
- Set the dashboard property to newly created instance of **$.ig.RVDashboard**

```
var revealView = new $.ig.RevealView("#revealView");
revealView.startInEditMode = true;
revealView.dashboard = new $.ig.RVDashboard();
```

You can find a working example, **CreateDashboard.cshtml**, in the *UpMedia* web application distributed with the SDK.

Editing & Saving Dashboards

Editing dashboards

The **dashboard** property (type $.ig.RVDashboard) of **revealView** is updated when the end user starts editing the dashboard. For example, when adding or removing visualizations or filters, $.ig.RVDashboard's collections get automatically updated.

In addition, the **$.ig.RVDashboard** class includes the **onHasChangesChanged** property that is very useful to check if there are unsaved changes in the dashboard.

Code Sample:

```
dashboard.onHasChangesChanged = function (hasChanges) {
   console.log("Has Changes: " + hasChanges);
};
```

After a user finishes editing a visualization, upon closing the Visualization Editor, the $.ig.RevealView's **visualizationEditorClosed** event is fired:

```
revealView.onVisualizationEditorClosed = function (args) {
    if (args.isCancelled) {
        console.log("Visualization editor cancelled " + (args.isNewVisualization ? "creating a new visualization " : "editing " + args.visualization.title));
        return;
    }
    if (args.isNewVisualization) {
        console.log("New Visualization created: " + args.visualization.title);
    } else {
        console.log("Visualization modified: " + args.visualization.title);
    }
 };
```

In the case that you need to control how to add new visualizations please refer to **Creating New Visualizations and Dashboards**.

Saving Dashboards

As described in **Loading Dashboard Files**, there are two ways to handle how you save changes to dashboards:

- **Client-side**: To use this method you need to set a function in the **onSave** attribute of the **revealView** object. This is the recommended approach as it gives more flexibility to the containing app on how operations (save and save as) are performed.

  Code Sample:

  ```
  revealView.onSave = function(rv, saveEvent) {
     saveEvent.serialize(function(blobValue) {
        //TODO: save the blob value, for example using a XMLHttpRequest object
        //to POST to the server
     });
  };
  ```

  In case you don't want to handle the save action, you can turn off the option to edit dashboards by setting:

  ```
  revealView.canSaveAs = false;
  ```

  This might be useful, for example, when your users are not supposed to make changes.

- **Server-side**: When the **onSave** event is not set in the **$.ig.RevealView** object, the default server-side saving method is used. After the end user saves a modified dashboard, a HTTP POST request is invoked. As a result, the **SaveDashboardAsync** method of the currently defined SDK context is invoked. And you get the _dashboardId as string and a Stream representation of the dashboard in

dashboardStream.

With the server-side approach, you only need to implement the code client-side but you lose flexibility client-side. This means, for example, that the user cannot select the final location where the dashboard will be stored. For further details about the SDK context, please refer to **Defining the Server Context.**.

Exporting a Dashboard or a Visualization

Overview

If you want to export a dashboard or a particular visualization, you can choose between the following export options:

- as an **image**;
- as a **PDF** document;
- as a **PowerPoint** presentation;
- into **Excel** data format.

To enable a dashboard or a visualization export, you can:

- use the export setting in the $.ig.RevealView, or
- initiate export programmatically outside of the $.ig.RevealView, when exporting **as an image**.

Prerequisites for Export as an Image Option

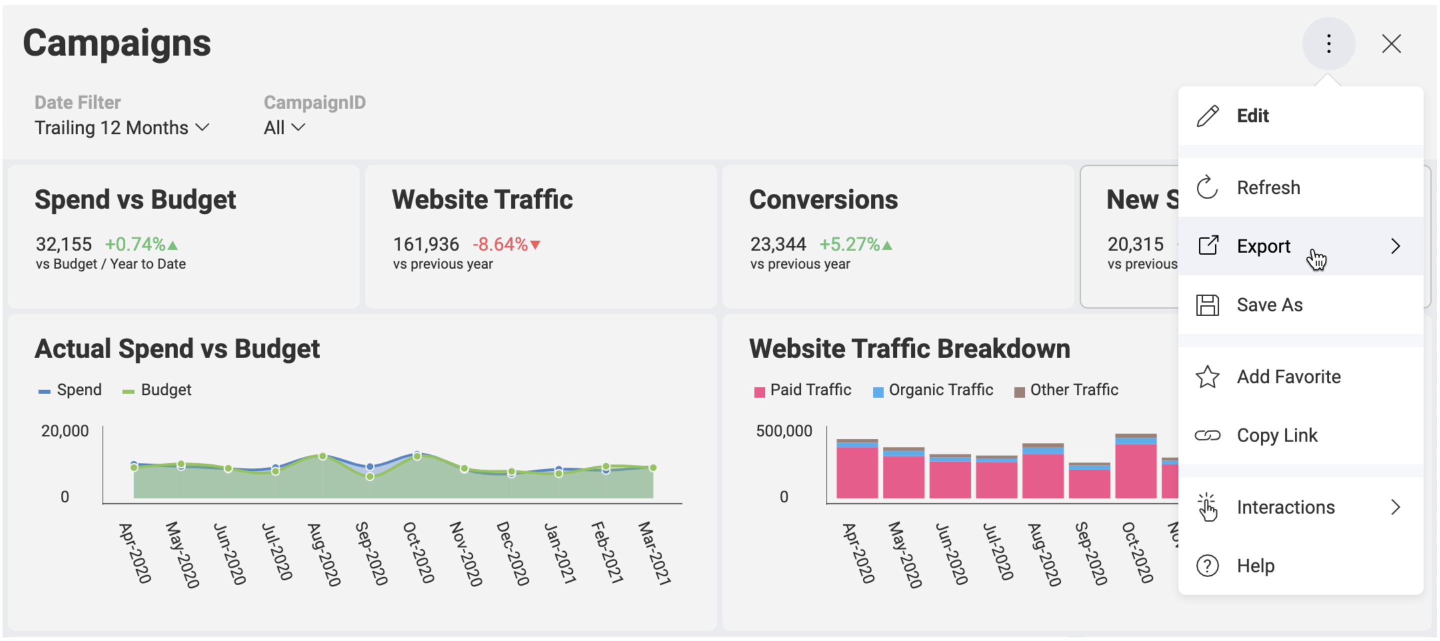
You need to enable the **export image** functionality in the server-side. To do this, please refer to Enabling server-side screenshot generation.

Using the Export Setting

To enable your end users to generate an image, document or a presentation out of a dashboard you simply need to set the relevant property to true:

- **revealView.showExportImage** - for export as an **image**;
- **revealView.showExportToPDF** - for export as a **PDF** document;
- **revealView.showExportToPowerpoint** - for export as a **PowerPoint** presentation;
- **revealView.showExportToExcel** - for export in **Excel** data format.

This will make the *Export* button available in the overflow menu when a dashboard is opened or a particular visualization is maximized.



When the user clicks the *Export* button, they can choose one of the enabled export options.

Specifics when using the image export option

If the user chooses the *Export Image* from the export options, the *Export image* dialog will open. Here, the user can choose between two options: *Copy to clipboard* and *Export Image*. If they click the *Export Image* button on the bottom right, the image will be sent to the end user.

In case the containing application needs to process the exported image in a different way it could provide **onImageExported** callback where the output image could be accessed. Here's a sample implementation of the onImageExported callback:

```
revealView.onImageExported = function (img) {
var body = window.open("about:blank").document.body;
body.appendChild(img);
}
```

Programmatically Initiated Image Export

To get an image of the $.ig.RevealView programmatically, you will need to invoke the **ToImage** method. Calling this method will not result in showing the *Export Image* dialog. This way, you can get a screenshot when the user clicks a button, which is outside of the $.ig.RevealView. This method will create a screenshot of the revealView component as it is displayed on the screen.

```
var image = revealView.toImage();
```

Keep in mind that if the end user has any dialog opened at the time of the *ToImage* method call, the dialog will appear in the screenshot together with the dashboard.

This could be useful in a scenario where the containing application would want to hide the expert button in the reveal view and trigger an export to image from another interaction that occurs outside of the reveal view.

Dashboard Linking

Overview

The Reveal application supports dashboard linking, which allows users to navigate through dashboards. By moving from dashboard to dashboard, you can go from a high level overview of the business' reality to a more detailed view with the specifics.

Common Use Cases

You can have, for example, a "Company 360" dashboard showing key performance indicators for each division (HR, Sales, Marketing). Once the user maximizes one of the visualizations, the navigation is triggered and the user is taken to another dashboard with more detailed information about that division.

Alternatively, you can also use dashboard linking to navigate to a more specific dashboard. For example, in a dashboard with a visualization showing the Top 25 Customers, by selecting one of the customers the user will navigate to a new dashboard. This new dashboard will display detailed information about the selected customer, like recent purchases, contact information, top selling products, etc.

For further details about the Dashboard Linking functionality, refer to **Dashboard Linking** from Reveal's User Guide.

Code Example

You can use dashboard linking with the SDK, but the containing application needs to be involved when the navigation is being triggered. As the SDK does not handle where dashboards are stored, it needs the containing application to provide a dashboard file for the target dashboard.

As a practical example, you can use the **Marketing.cshtml** page in the *UpMedia* sample distributed with the SDK.

Basically, you just have to do two things and the SDK will take care of the rest:

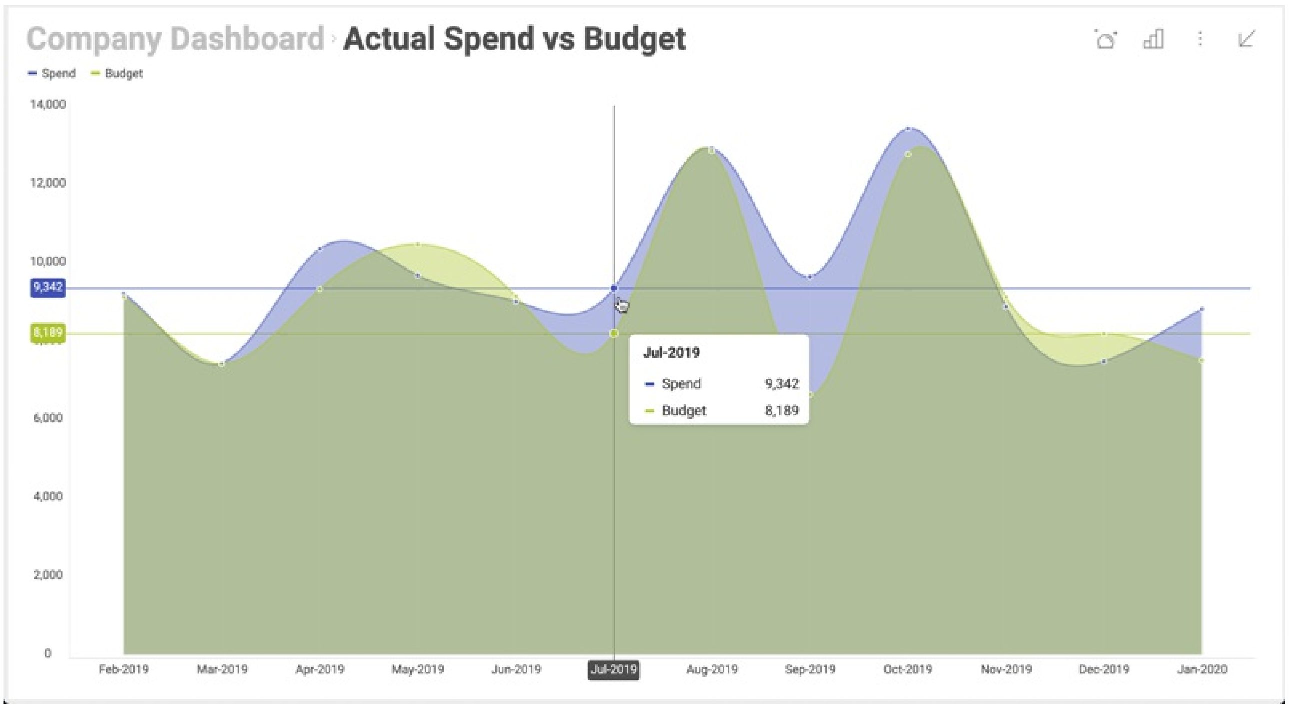
1. Handle the **onVisualizationLinkingDashboard** event.
2. Invoke the callback with the ID of the target dashboard.

```
revealView.onVisualizationLinkingDashboard = function (title, url, callback) {
    //provide the dashboard id of the target of the link
    callback("Campaigns");
};
```

Maximizing Visualizations and Single Visualization Mode

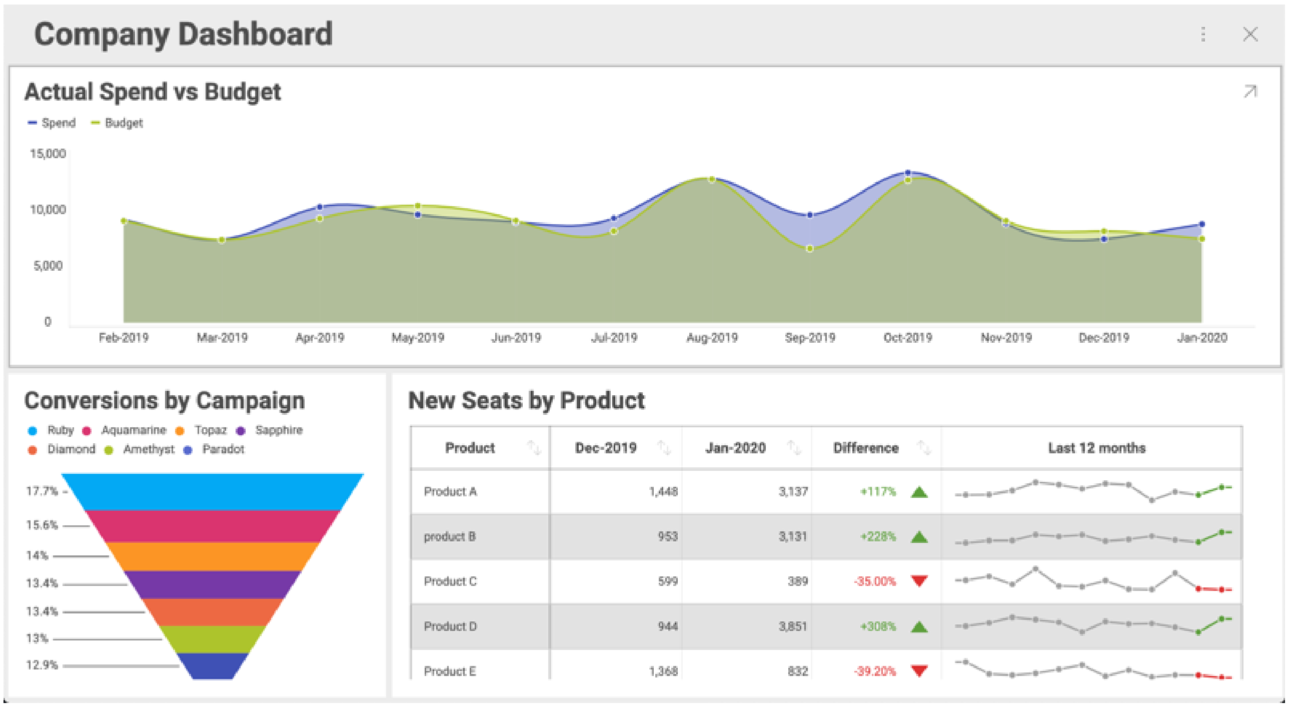
Overview

When displaying a dashboard to the user, there are some cases in which you'd like to display just one maximized visualization. In addition, you might also want to lock the initial visualization and prevent the user from accessing the whole dashboard. You can achieve both scenarios using the Web Client SDK.



Example Details

Let's assume that you have a dashboard with three visualizations, where each visualization is showing data for a different division of your company, i.e., "Marketing", "Sales" and "HR".



Product	Dec-2019	Jan-2020	Difference	Last 12 months
Product A	1,448	3,137	+117% ▲	
product B	953	3,131	+228% ▲	
Product C	599	389	-35.00% ▼	
Product D	944	3,851	+308% ▲	
Product E	1,368	832	-39.20% ▼	

In this example, you'd like to showcase these visualizations in your corporate application. You want to include them as part of the information displayed on each division's home page.

Maximizing Visualizations

To open a dashboard with a maximized visualization, you need to set the dashboard property of **revealView** first. Then, set the **maximizedVisualization** property by passing the visualization you want maximized to the **$.ig.RevealView** instance. When you don't set a visualization in this attribute, the whole dashboard is displayed.

As shown in **Configuring the $.ig.RevealView object**, you can display a specific dashboard in your page. This time, you also need to set the **maximizedVisualization** property. As shown in the code snippet below with the visualization "Sales" from the dashboard with ID "AllDivisions".

```
<script type="text/javascript">
...

var dashboardId = 'AllDivisions';

$.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {
    var revealView = new $.ig.RevealView("#revealView");
    revealView.dashboard = dashboard;
    revealView.maximizedVisualization = dashboard.visualizations.getByTitle('Sales');

}, function (error) {
    console.log(error);
});
</script>

<div id="revealView" style="height:500px;" />
```

Although the initial maximized visualization will be the one with title 'Sales', the end user can still return to the dashboard and see the rest of the visualizations.

Single Visualization Mode

You may also want to lock the initial visualization, making it the only one displayed at all times. This way the dashboard works like a single visualization dashboard. This is the concept behind "single visualization mode".

To turn on the "single visualization mode", just set the **singleVisualizationMode** to true as shown below.

```
revealView.singleVisualizationMode = true;
```

After adding this single line, the dashboard will work as a single visualization dashboard. You can do the same for each division's home page, just replace the title of the visualization in dashboard.visualizations.getByTitle() with the right one.

Dynamically changing a locked visualization

It is also possible for you to dynamically change the single visualization being displayed, without reloading the page. From the user's perspective, your app would be a single page application with a selector of divisions and a maximized visualization. After the user chooses one division from the list, the maximized visualization is updated.

You can achieve this scenario by using the **maximizeVisualization** method in **$.ig.RevealView**, as shown below:

```
<script type="text/javascript">
    var dashboardId = 'AllDivisions';

    $.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {
        var revealView = window.revealView = new $.ig.RevealView("#revealView");
        revealView.singleVisualizationMode = true;
        revealView.dashboard = dashboard;
        revealView.maximizedVisualization = dashboard.visualizations.getByTitle('Sales');

    }, function (error) {
        console.log(error);
    });
    function maximizeVisualization(title) {
        var dashboard = window.revealView.dashboard;
        window.revealView.maximizedVisualization = dashboard.visualizations.getByTitle(title);
    }
</script>

<section style="display:grid;grid-template-rows:30px auto;">
    <section style="display:grid;grid-template-columns:auto auto auto;">
        <button onclick="maximizeVisualization('Sales')">Sales</button>
        <button onclick="maximizeVisualization('HR')">HR</button>
        <button onclick="maximizeVisualization('Marketing')">Marketing</button>
    </section>
    <div id="revealView" style="height:500px;" />
</section>
```

To take into account:

- The **$.ig.RevealView** object is set in _window.revealView</emphasis> in order to use it later when **maximizeVisualization** property is set.
- The buttons added to the section before the div are used just as an example. They were added as a means to switch the maximized visualization, in your case you'll have to to use a similar code in your application.
- In this example, the buttons are hardcoded to match the visualizations in the sample dashboard, but you can also generate the list of buttons dynamically by iterating the list of visualizations in the dashboard. For further details see **$.ig.RVDashboard.visualizations**.

Creating Custom Themes

Overview

When embedding analytics into your existing applications it is key that those dashboards match your app's look and feel. That's why you have full control over the Reveal dashboards through our SDK.

Key customizations you can achieve with custom themes:

- **Color palettes**: The colors used to show the series in your visualizations. You can add an unlimited number of colors. Once all colors are used in a visualization, Reveal will autogenerate new shades of these colors. This way your colors won't repeat and each value will have its own color.
- **Accent color**: The default accent color in Reveal is a shade of Blue that you can find in the **+ Dashboard** button and other interactive actions. You can change the color to match the same accent color you use in your applications.
- **Conditional formatting colors**: Change the default colors of the bounds you can set when using conditional formatting.
- **Font**: Reveal uses three types of text in the application: regular, medium and bold. You can specify the font uses for each of these text groups.
- **Visualization and dashboard background colors**: You can configure separately the background color of your dashboard and the background color of the visualizations.

Common Use Case: A New Custom Theme

Creating your own theme in Reveal is as easy as creating an instance of the new **$.ig.RevealTheme()** class. This class contains all the customizable settings listed in the overview.

When creating a new **$.ig.RevealTheme** instance, you will get the default values for each setting and you can modify them as needed.

Then, pass the theme instance to the **$.ig.RevealSdkSettings**'s class static theme property. If you have a dashboard or another Reveal component already displayed on your screen, you will need to render it again(set the dashboard property again) in order to see the applied changes.

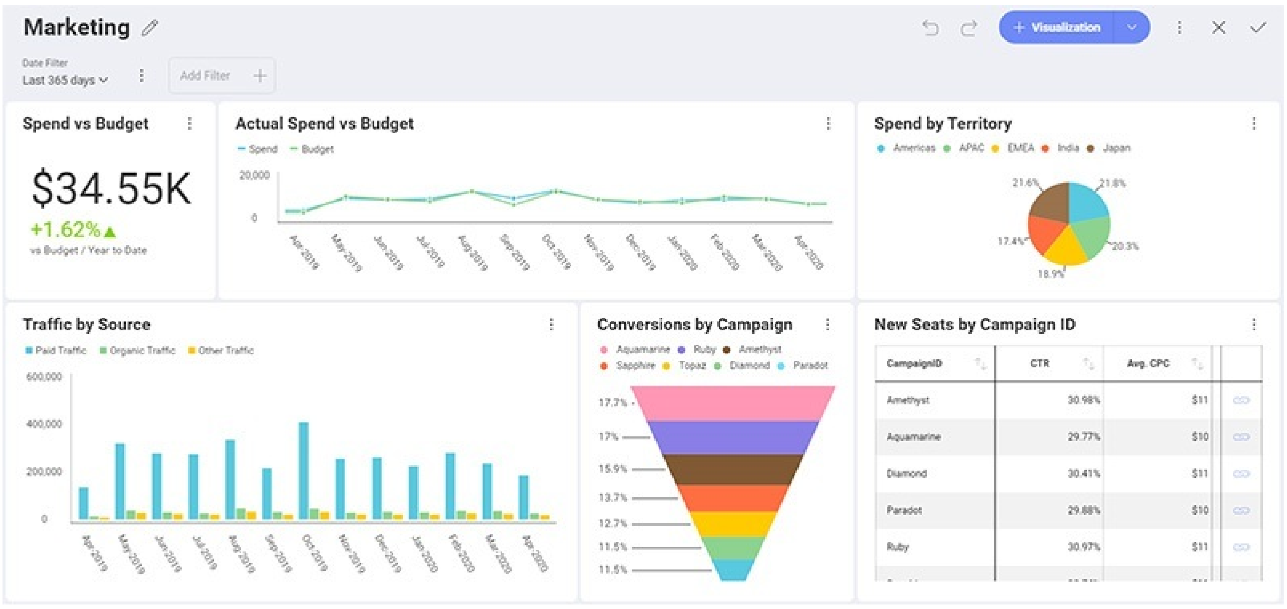
Common Use Case: Modifying a Custom Theme

You may have already applied your own theme but want to modify some of the settings without losing the changes you made to the others.

In this case, you need to get the theme static property from the **$.ig.RevealSdkSettings**. This enables you to get the last values you have set for your RevealTheme settings. Unlike the case when you create a new instance of the RevealTheme from scratch, after applying your changes and updating your theme again, you will get the current values for each setting you didn't modify instead of the default values.

Code Example

First, here's a sample dashboard before we make any changes:



CampaignID	CTR	Avg. CPC
Amethyst	30.98%	$11
Aquamarine	29.77%	$10
Diamond	30.41%	$11
Paradot	29.88%	$10
Ruby	30.97%	$11

In the following code snippet you can see how to create a new instance of the *revealTheme* class, apply the changes to the settings you want and update the theme in Reveal Web.

```
var revealTheme = new $.ig.RevealTheme();
revealTheme.chartColors = ["rgb(192, 80, 77)", "rgb(101, 197, 235)", "rgb(232, 77, 137)"];

revealTheme.mediumFont = "Gabriola";
revealTheme.boldFont = "Comic Sans MS";
revealTheme.fontColor = "rgb(31, 59, 84)";
revealTheme.accentColor = "rgb(192, 80, 77)";
revealTheme.dashboardBackgroundColor = "rgb(232, 235, 252)";

$.ig.RevealSdkSettings.theme = revealTheme;
```

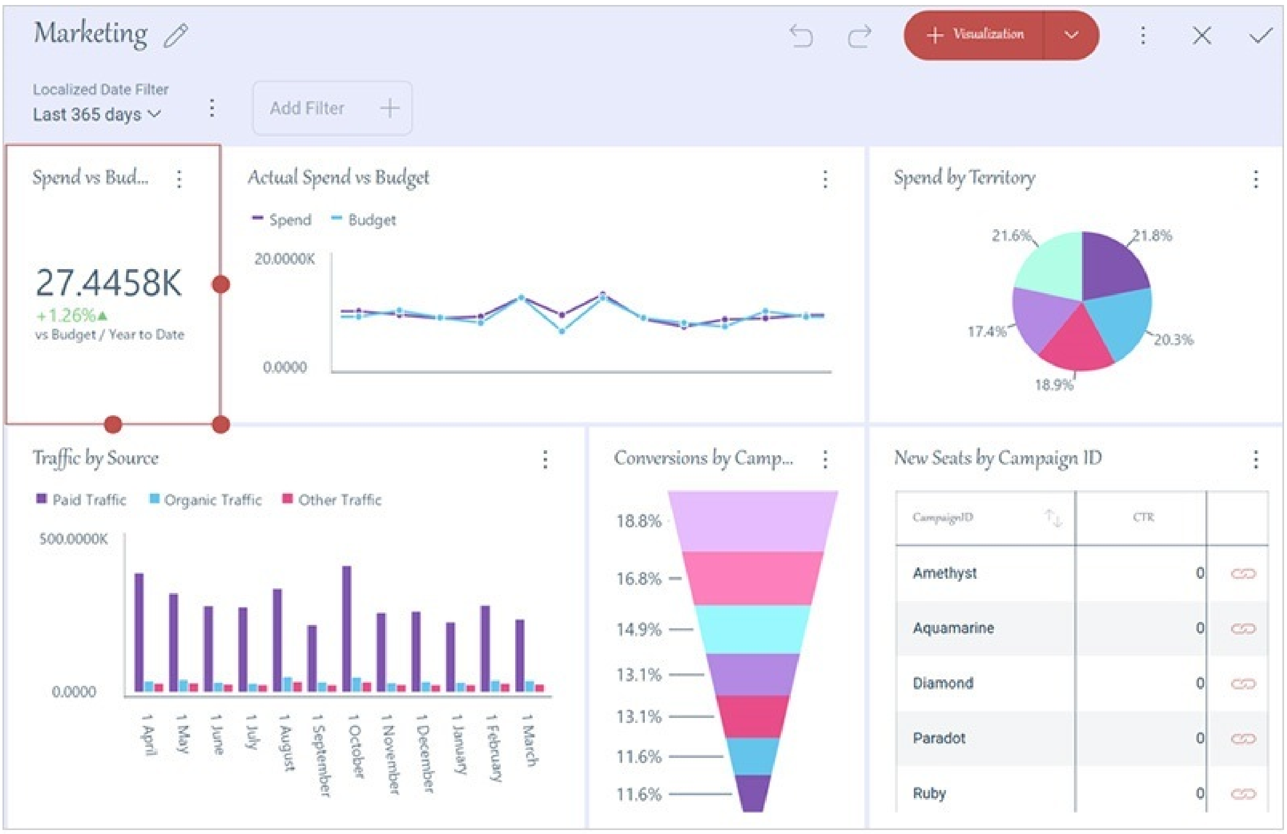
Note

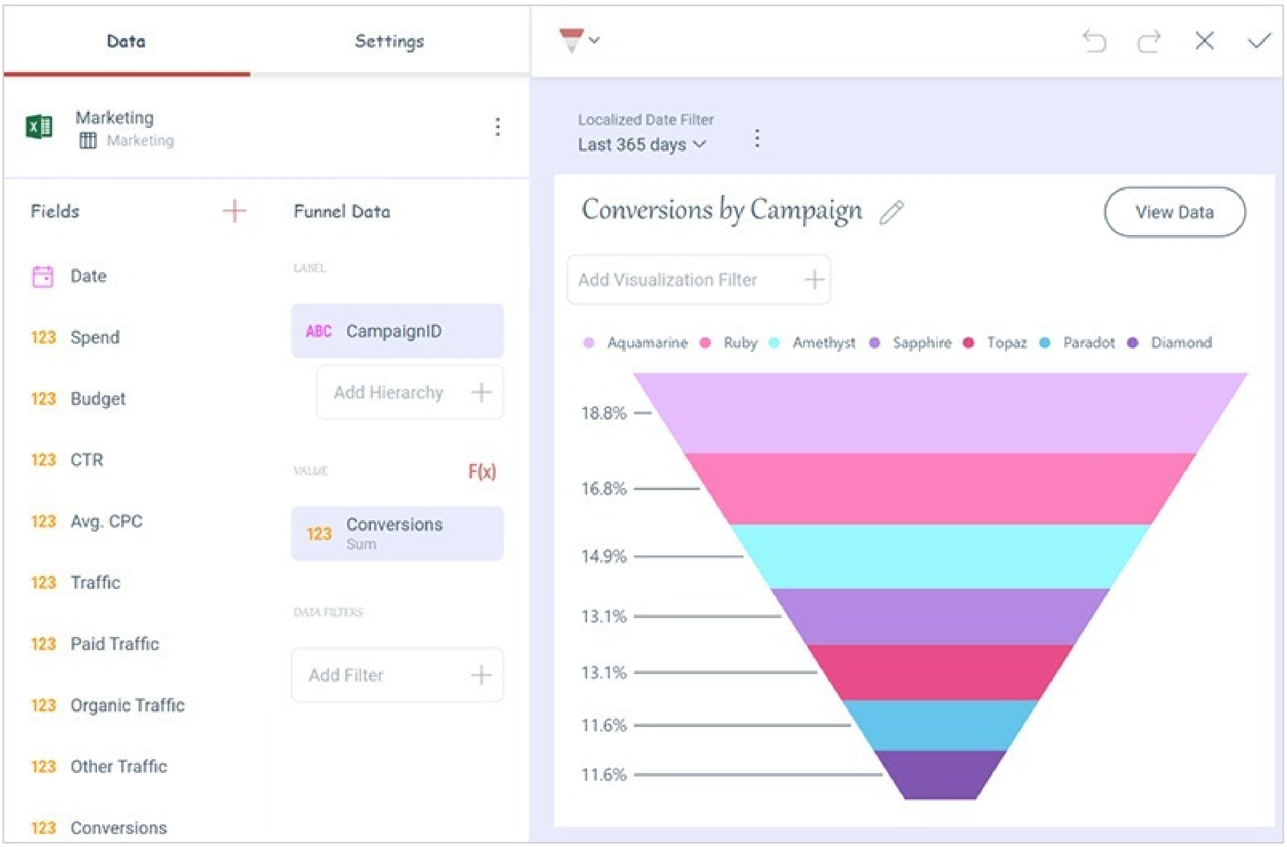
When defining the boldFont, regularFont or mediumFont settings of a reveal theme you need to pass the exact font family name. The weight is defined by the definition of the font itself and not in the name. You might need to use @font-face (CSS property) to make sure the font-face name specified in the $.ig.RevealTheme font settings are available.

In addition, for the fonts customization you need to add these lines to the CSS of the page:

```
<link href="https://fonts.googleapis.com/css?family=Righteous" rel="stylesheet">
<link href="https://fonts.googleapis.com/css?family=Domine" rel="stylesheet">
<link href="https://fonts.googleapis.com/css?family=Caveat" rel="stylesheet">
```

After implementing the theme changes, below you can see the results for both the Dashboard and Visualization Editors.





Using Color Types

You can use either RGB (red, green, blue) or HEX colors to specify the color settings.

```
revealTheme.dashboardBackgroundColor = "rgb(232, 235, 252)";
revealTheme.dashboardBackgroundColor = "#E8EBFC";
```

Built-In Themes

Reveal SDK comes with four pre-built themes: *Mountain Light*, *Mountain Dark*, *Ocean Light*, and *Ocean Dark*. You can set the one that best matches your application's design, or you can also use it as the basis for your custom theme modifications.

Apply the settings of a chosen pre-built theme by using the *UpdateCurrentTheme* method.

Mountain Light Theme

```
$.ig.RevealSdkSettings.theme = new $.ig.MountainLightTheme();
```

Note

Mountain Light contains the default values for the customizable theme settings. This means Mountain Light and the Reveal Theme look basically the same way.

Mountain Dark Theme

```
$.ig.RevealSdkSettings.theme = new $.ig.MountainDarkTheme();
```

Ocean Light Theme

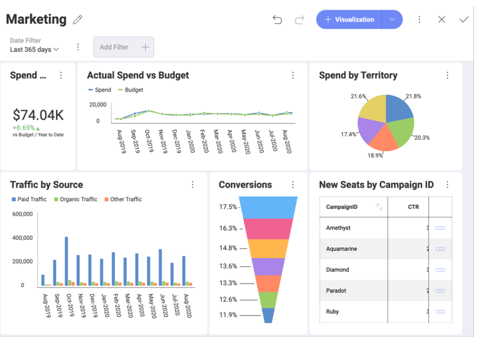
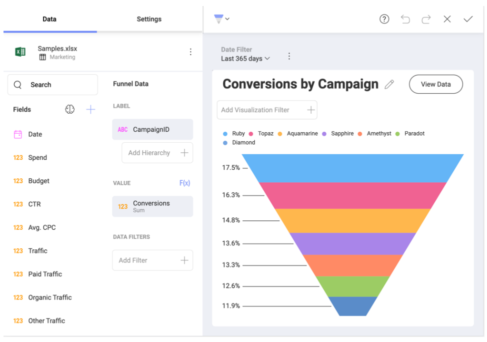
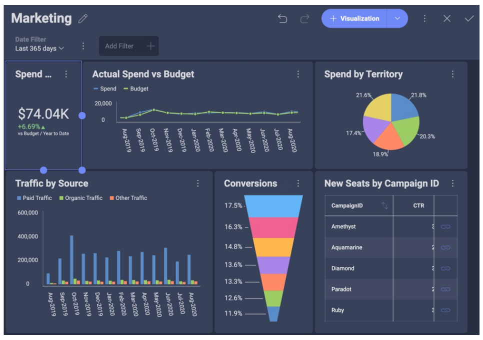
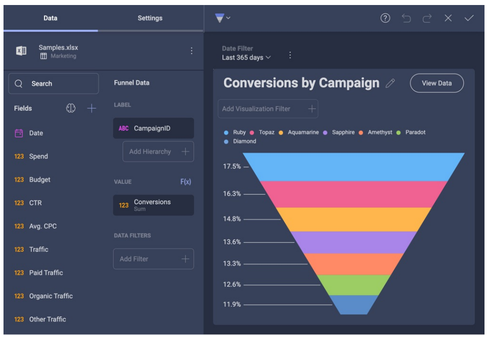
```
$.ig.RevealSdkSettings.theme = new $.ig.OceanLightTheme();
```

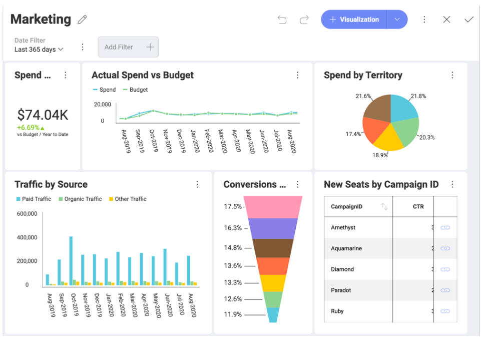
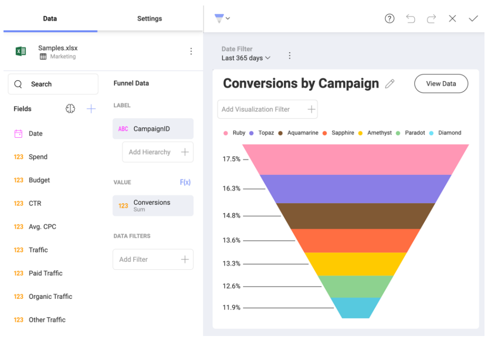
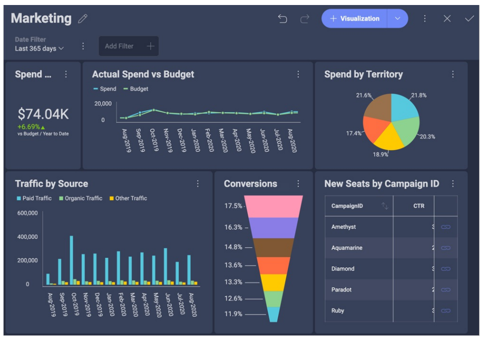
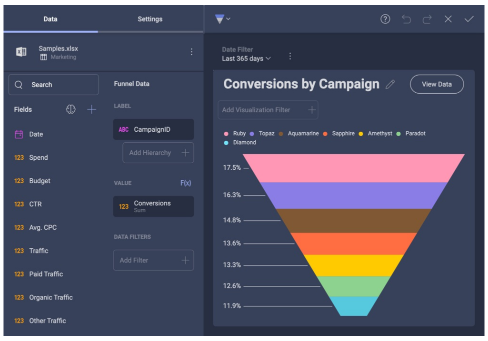
Ocean Dark Theme

```
$.ig.RevealSdkSettings.theme = new $.ig.OceanDarkTheme();
```

How the Built-In Themes Look?

Below, you will find a table showing how the *Visualization Editor* and *Dashboard Editor* look when each of the pre-built themes is applied.

THEME	DASHBOARD EDITOR	VISUALIZATION EDITOR
Mountain Light (Default)	 Marketing Spend ... $74.04K Actual Spend vs Budget Spend by Territory Traffic by Source Conversions New Seats by Campaign ID	 Data Settings Samples.xlsx Conversions by Campaign View Data
Mountain Dark	 Marketing Spend ... $74.04K Actual Spend vs Budget Spend by Territory Traffic by Source Conversions New Seats by Campaign ID	 Data Settings Samples.xlsx Conversions by Campaign View Data

THEME	DASHBOARD EDITOR	VISUALIZATION EDITOR
Ocean Light		
Ocean Dark		

Handling User Click Events

Overview

The SDK allows you to handle when the user clicks a cell with data within a visualization. This is very useful, for example, to provide your own navigation, to change existing selections in your app, among others.

Code

You can handle user click events by registering to the **onVisualizationDataPointClicked** event:

```
window.revealView.onVisualizationDataPointClicked = function (visualization, cell, row) {
    alert('Visualization clicked: ' + visualization.title() + ", cell: " + cell.value);
};
```

In the callback function you receive information about the location of the click:

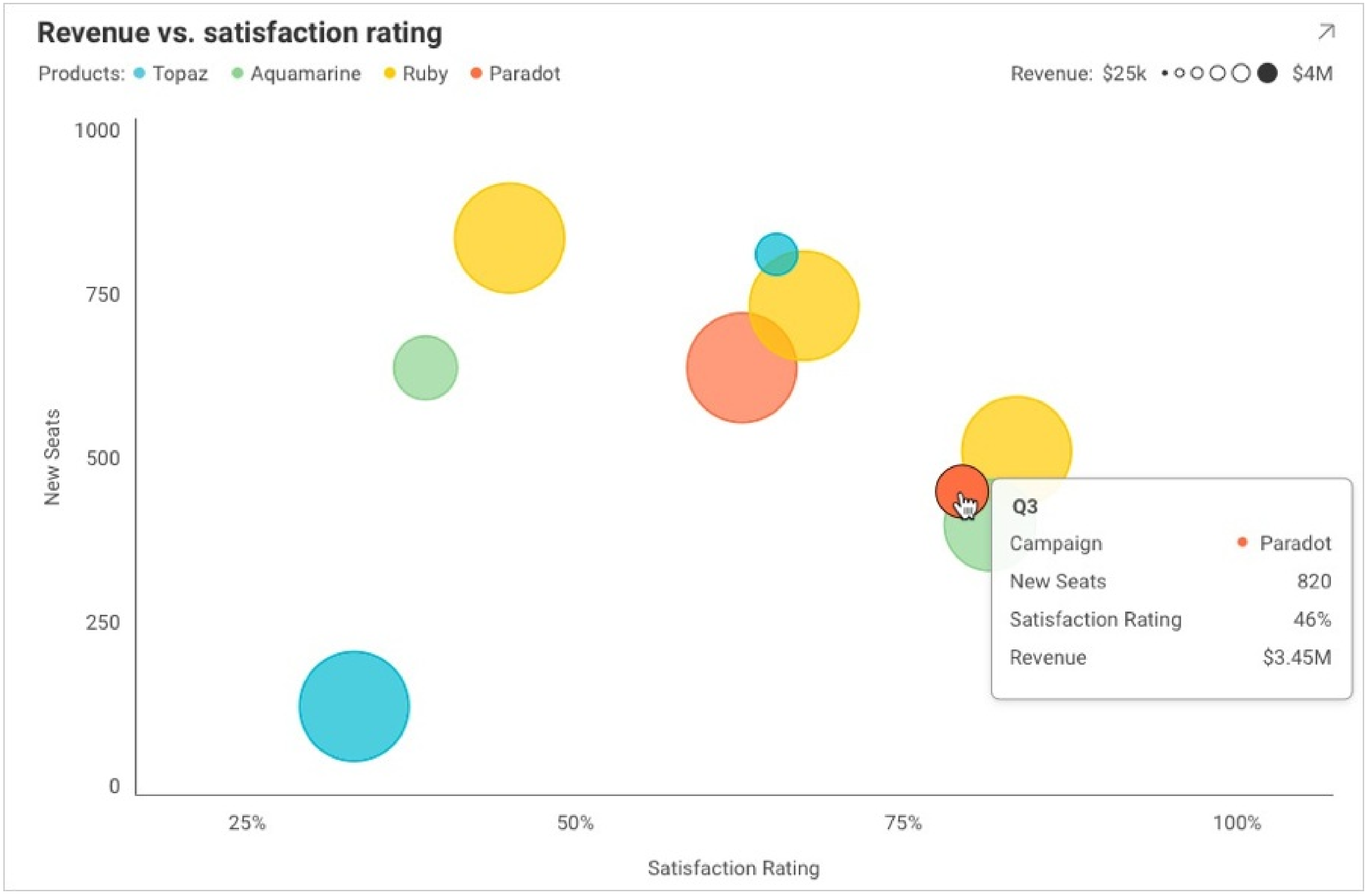
- the name of the visualization clicked;
- the values for the cell clicked (including value, formatted value, and the column's name);
- the rest of the values in the same cell.

You can use the rest of the values in the cell to search a specific attribute, like customer ID if you want to change the selected customer in your application. Doesn't matter that the user clicked another cell, like the sales amount for that customer, you'll still get the information you need.

Working with Tooltips

Overview

There is an event that is triggered whenever the end-user hovers over a series in a visualization or clicks on the series (as shown in the image below). This event called **.onTooltipShowing**, gives you more flexibility regarding how you show Tooltips in your visualizations.



Common Use Cases

You can cancel the Tooltip event or modify what is shown to the user. Most common examples include:

- You want to disable tooltips altogether or only show them for specific visualizations.
- You want to display data in the tooltip that is outside of the RevealView component that might be more valuable to your viewers.

Please note that this event will not be triggered for visualizations that do not support Tooltips, such as grids, gauges, and others.

Code Example

In the following code snippet, you can see how to disable tooltips for a visualization and still get additional information from the event arguments when the end-user hovers over or clicks on this visualization.

```
revealView.onTooltipShowing = function (args) {
    if (args.visualization.title == "NoNeedForTooltips") {
            args.Cancel = true;
    }
    console.log("onTooltipShowing: visualization: " + args.visualization.title() + ",cell: " + args.cell.value + ", row:" + args.row.length);
};
```

The event arguments include information about the visualization that is being hovered over or clicked on, the exact cell of data hovered over or clicked, the whole row of this cell (in case you need information from other columns), and, of course, the Cancel boolean.

Showing/Hiding User Interface Elements

The **$.ig.RevealView** component can be used to enable or disable different features and/or UI elements towards the end user. Many of the available properties are of the Boolean type and can be very straightforward to use, but others not so much.

The *revealView* instance and the DOM element created below are assumed by all the code snippets in this topic:

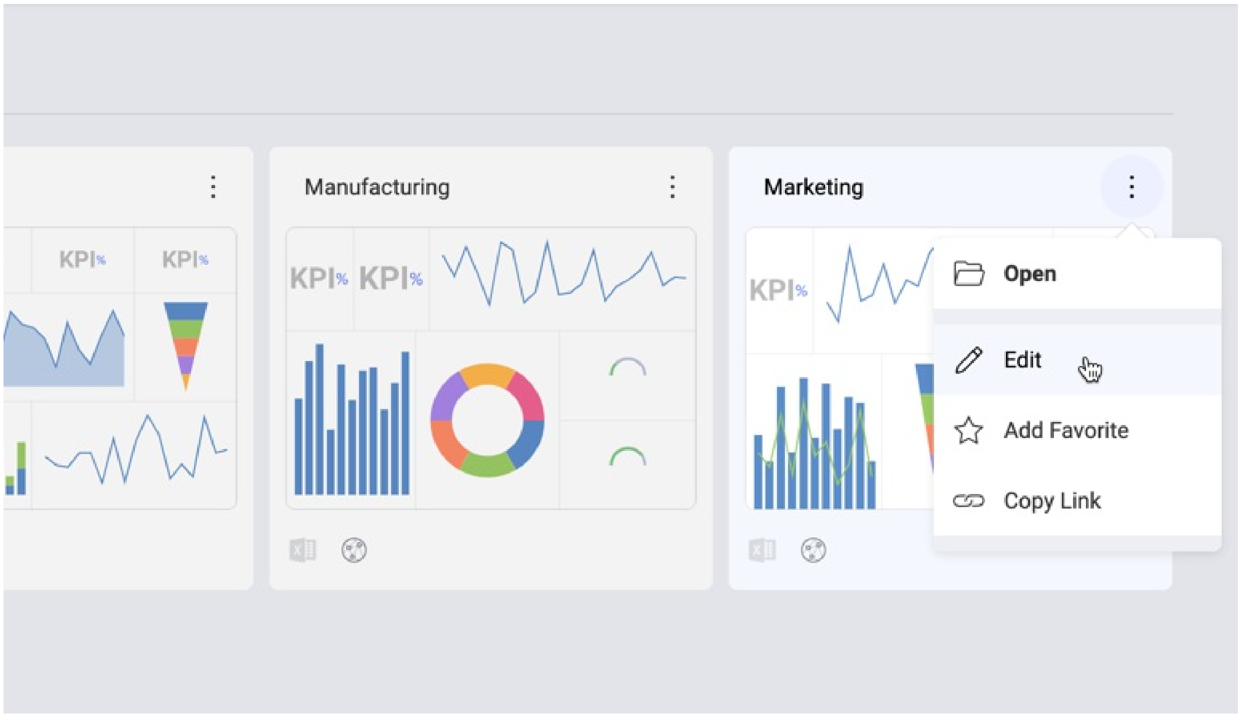
```
var revealView = new $.ig.RevealView("#revealView");
...
<div id="revealView" style="height:500px;" />
```

Note

Depending on your css layout approach you might need the element hosting the RevealView to be "positioned" by setting a position attribute that is not static (like relative or absolute).

canEdit

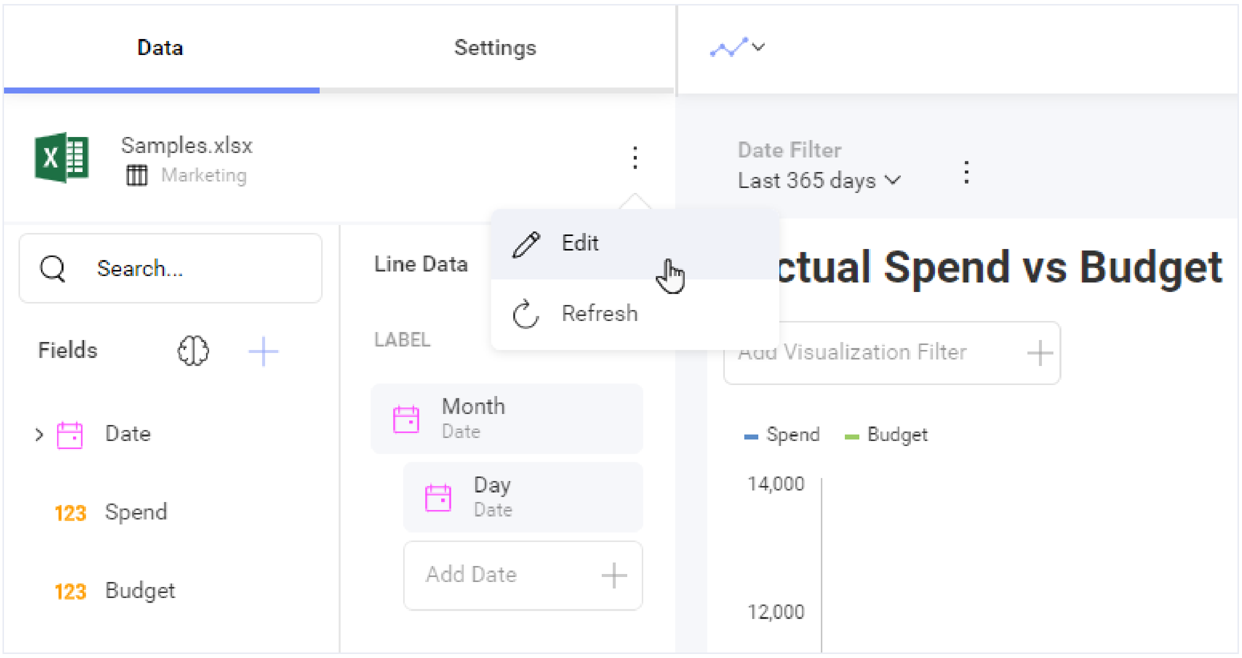
This property can be used to disable the user's ability to edit dashboards.



```
revealView.canEdit = false;
```

showEditDataSource

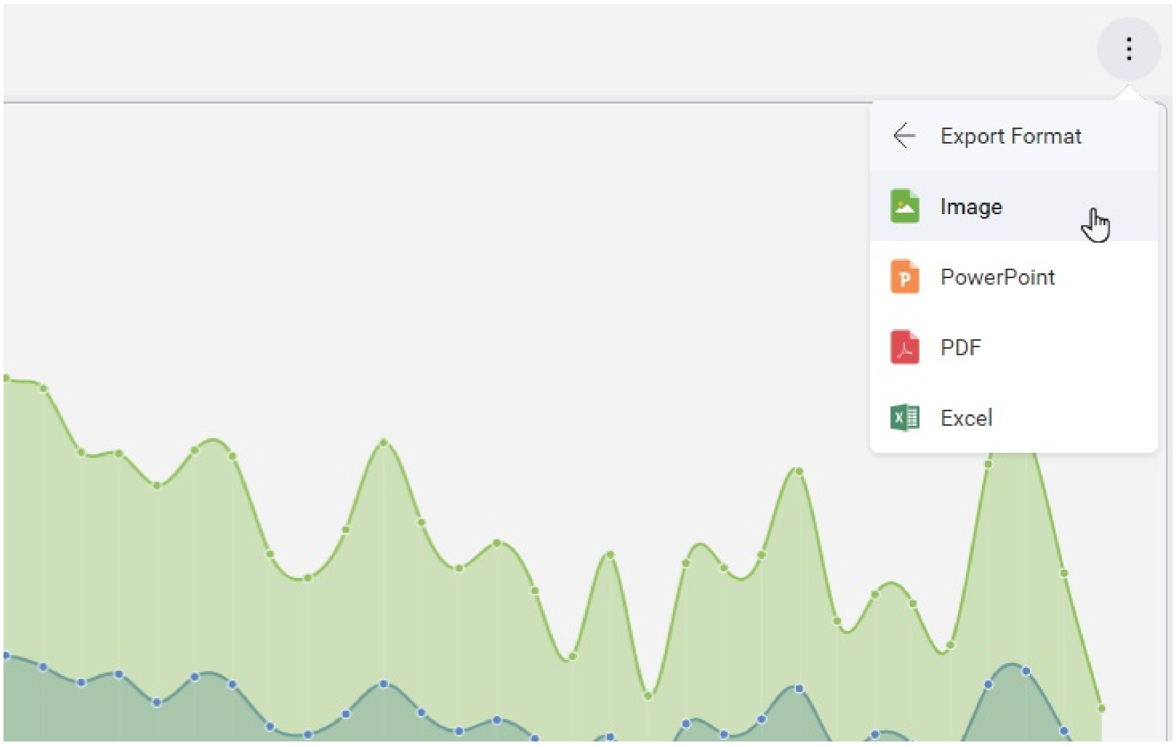
This property can be used to disable the editing of a dashboard datasource.



```
revealView.showEditDataSource = false;
```

showExportImage

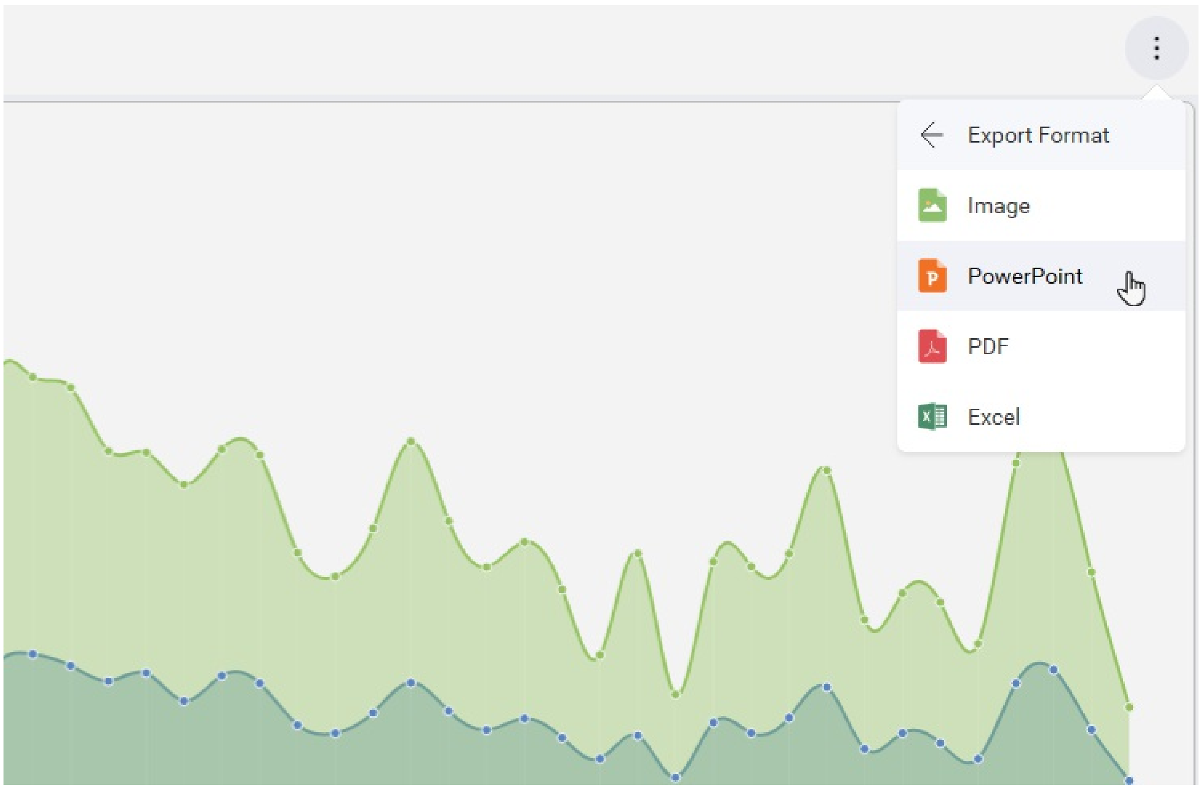
This property can be used to disable exporting the dashboad to an image.



```
revealView.showExportImage = false;
```

showExportToPowerpoint

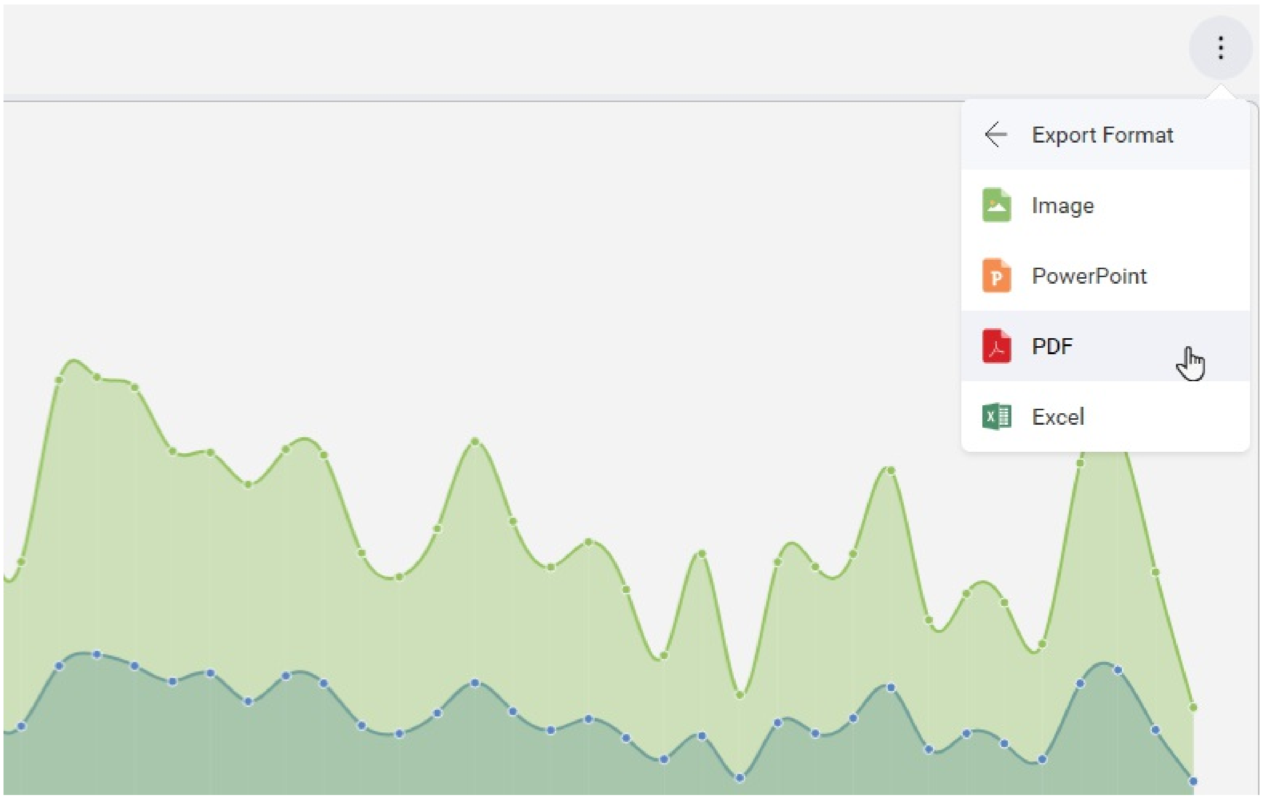
This property can be used to disable exporting the dashboad to PowerPoint.



```
revealView.showExportToPowerpoint = false;
```

showExportToPDF

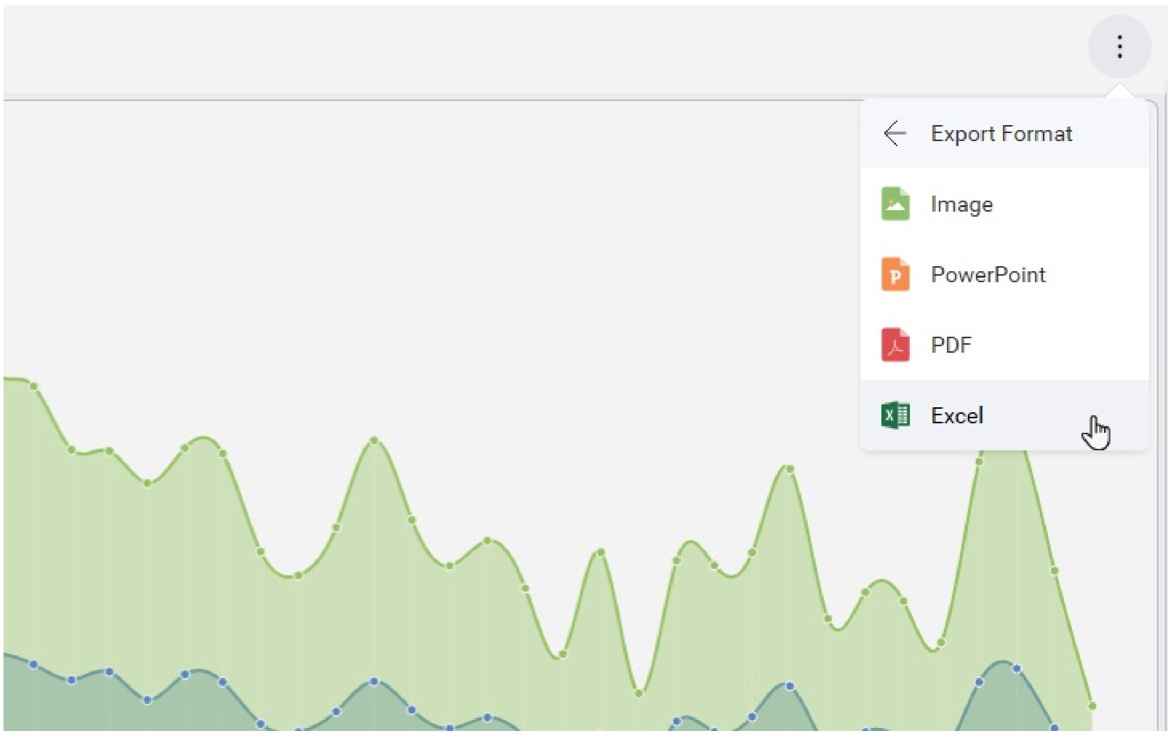
This property can be used to disable exporting the dashboad to PDF.



```
revealView.showExportToPDF = false;
```

showExportToExcel

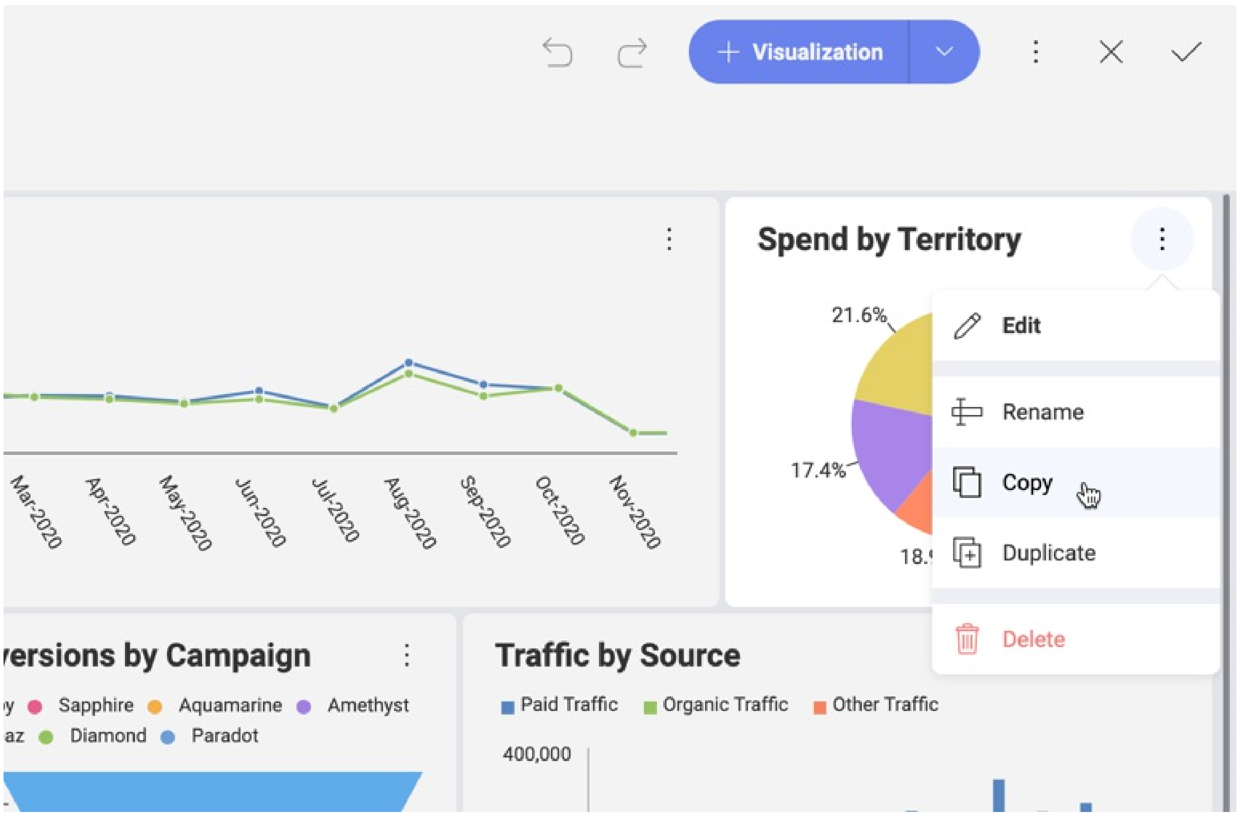
This property can be used to disable exporting the dashboad to Excel.



```
revealView.showExportToExcel = false;
```

canCopyVisualization

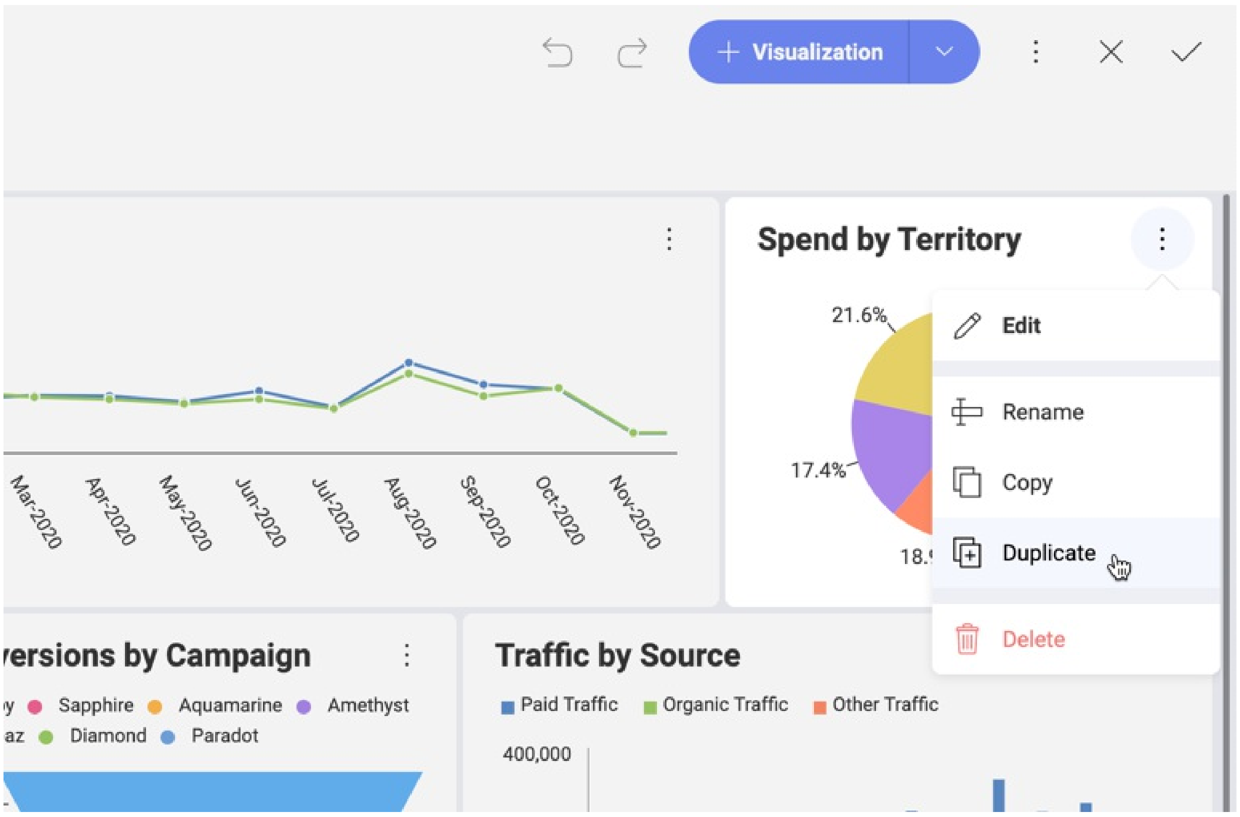
This property can be used to disable the ability to copy a visualization and later paste it in the current dashboard or a different one.



```
revealView.canCopyVisualization = false;
```

canDuplicateVisualization

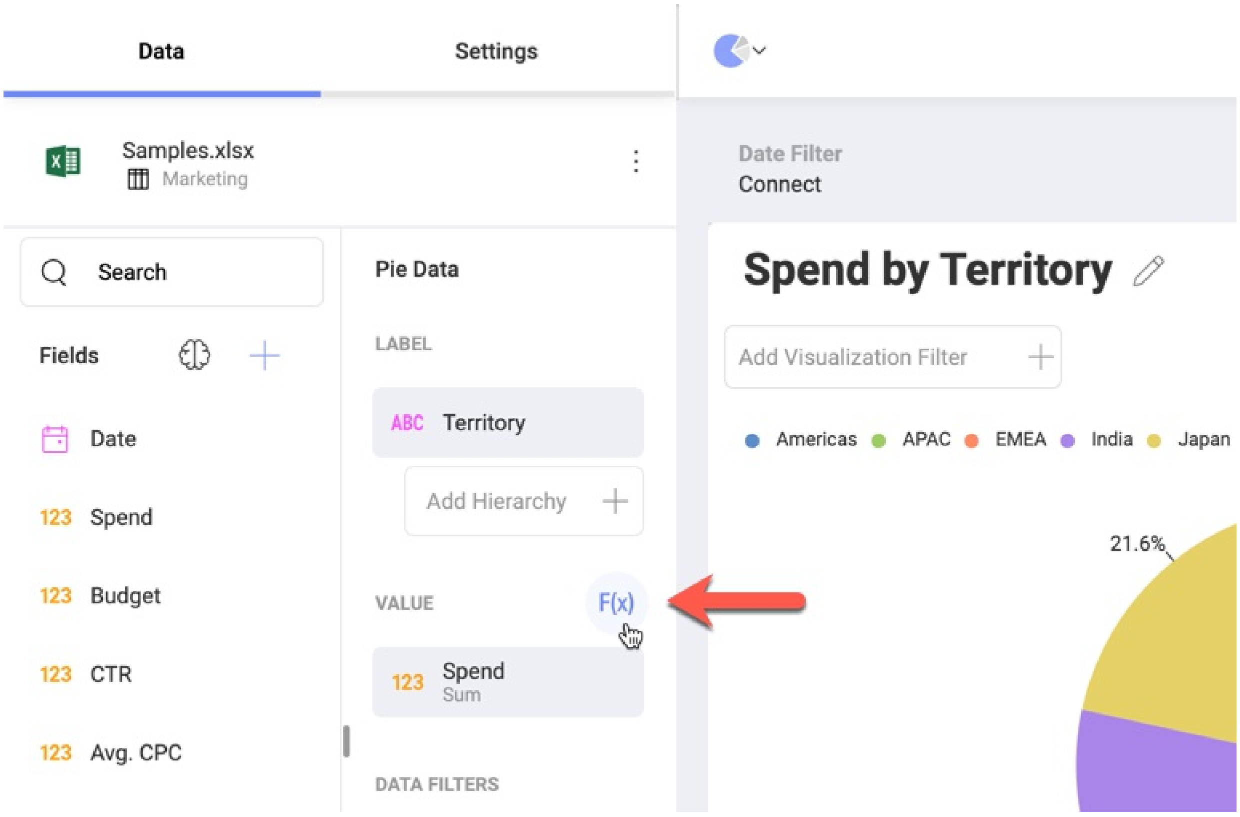
This property can be used to disable the ability to duplicate a visualization in the current dashboard.



```
revealView.canDuplicateVisualization = false;
```

canAddPostCalculatedFields

This property can be used to disable the ability to add a new post-calculated field in the current dashboard.

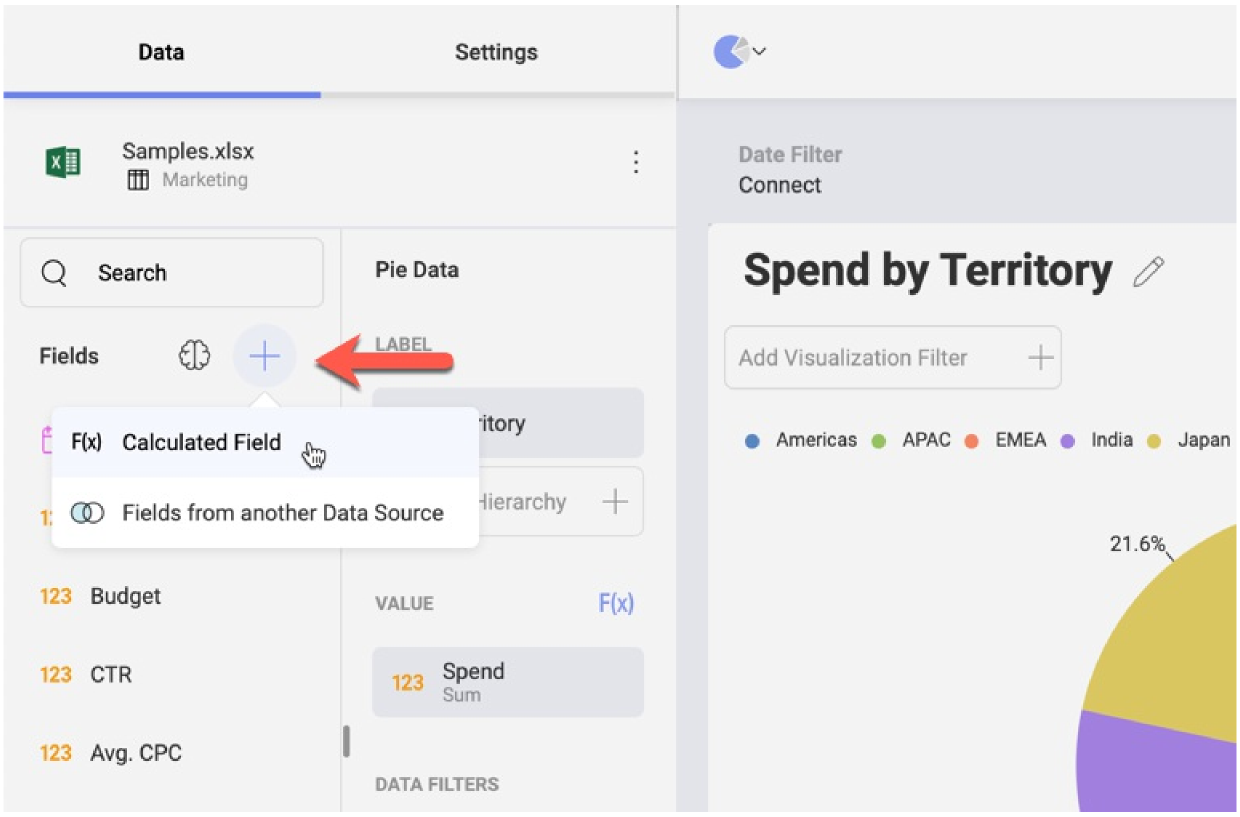


Post-calculated fields are new fields in the data set and are created by applying a formula on already summarized values. For further details, please refer to the Reveal Help.

```
revealView.canAddPostCalculatedFields = false;
```

canAddCalculatedFields

This property can be used to disable the ability to add a new pre-calculated field in the current dashboard.

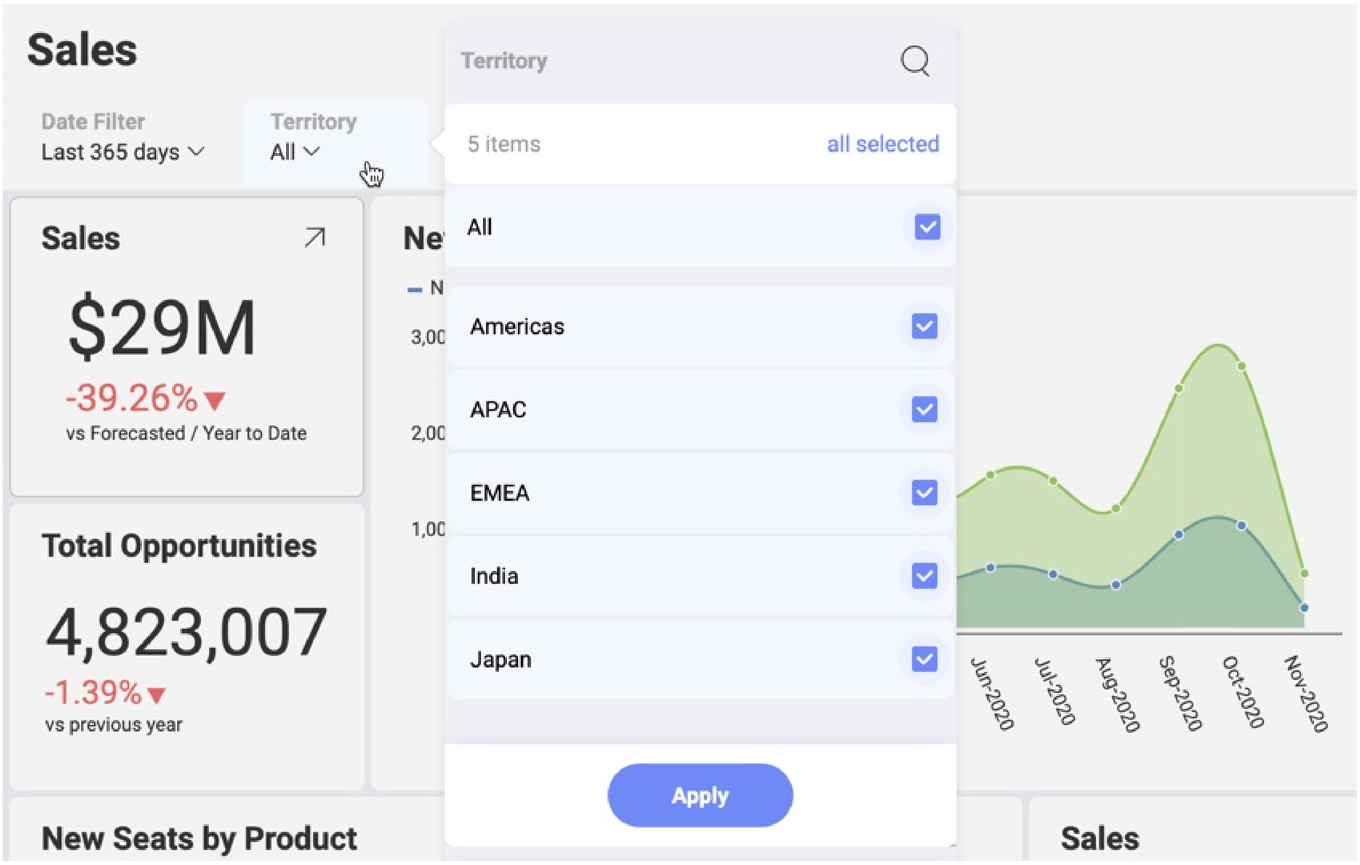


Pre-calculated fields are new fields in the data set and are evaluated before executing data editor aggregations. For further details, please refer to the Reveal Help.

```
revealView.canAddCalculatedFields = true;
```

showFilters

This property can be used to show or hide the Dashboard Filters UI to the user.

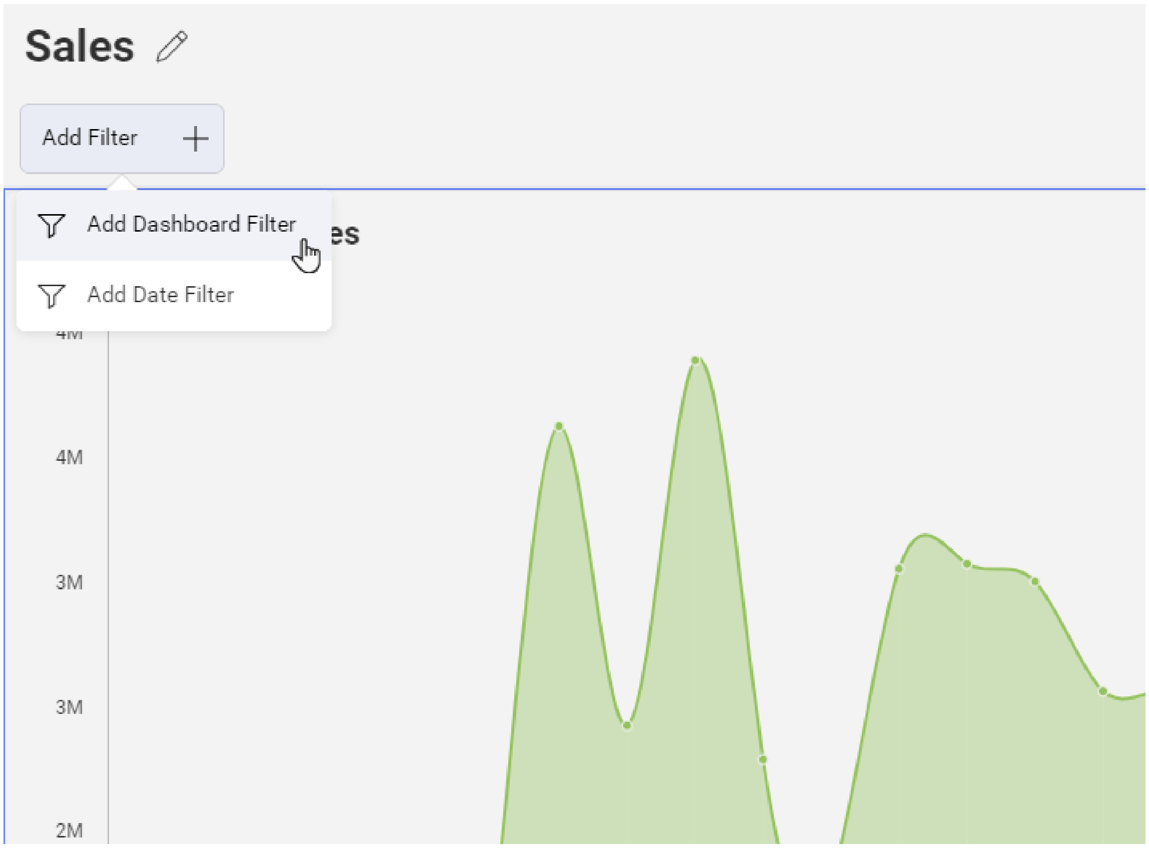


Dashboard filters allow you to slice the contents of the visualizations in a dashboard, all at once.

```
revealView.showFilters = true;
```

canAddDashboardFilter

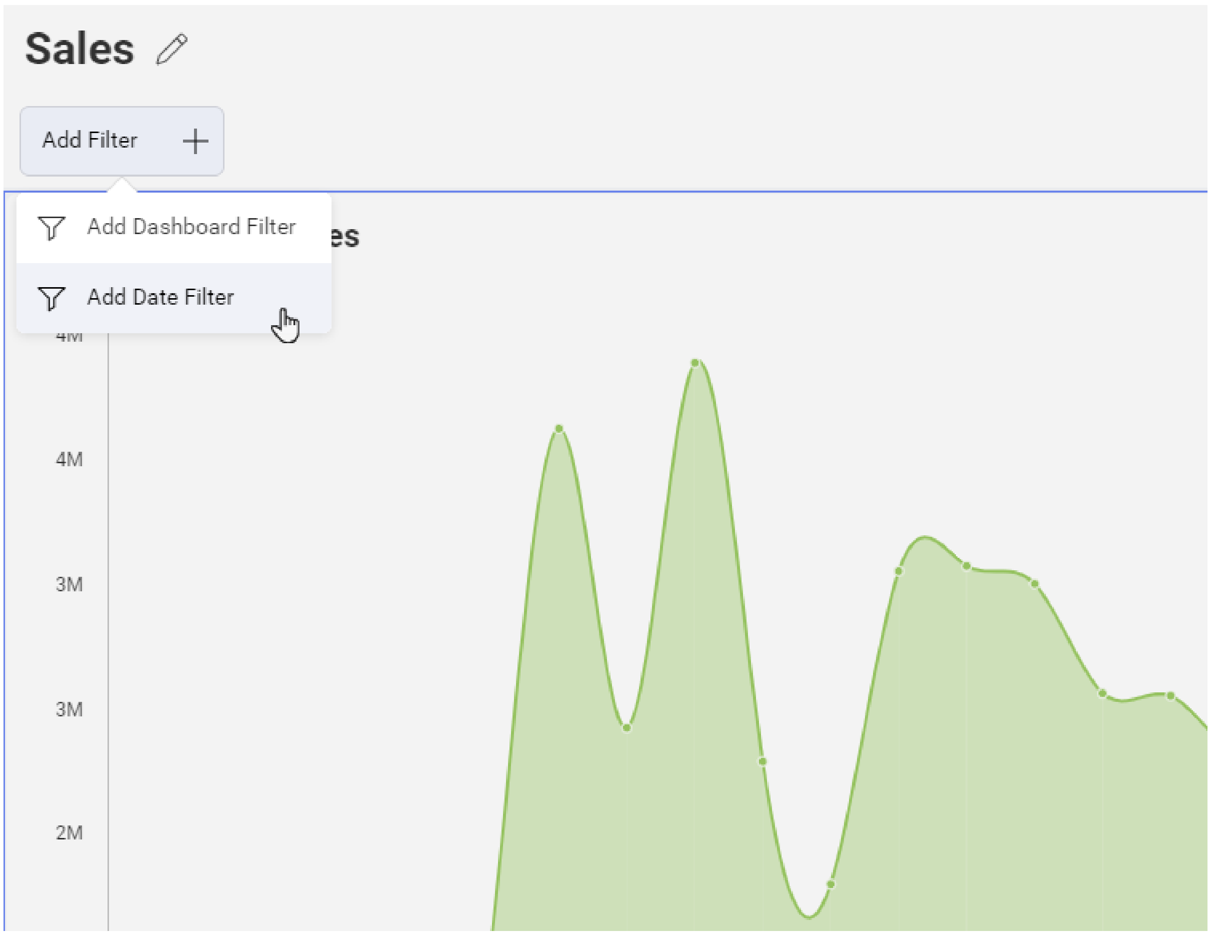
This property can be used to show or hide the Add Dashboard Filter menu item.



```
revealView.canAddDashboardFilter = false;
```

canAddDateFilter

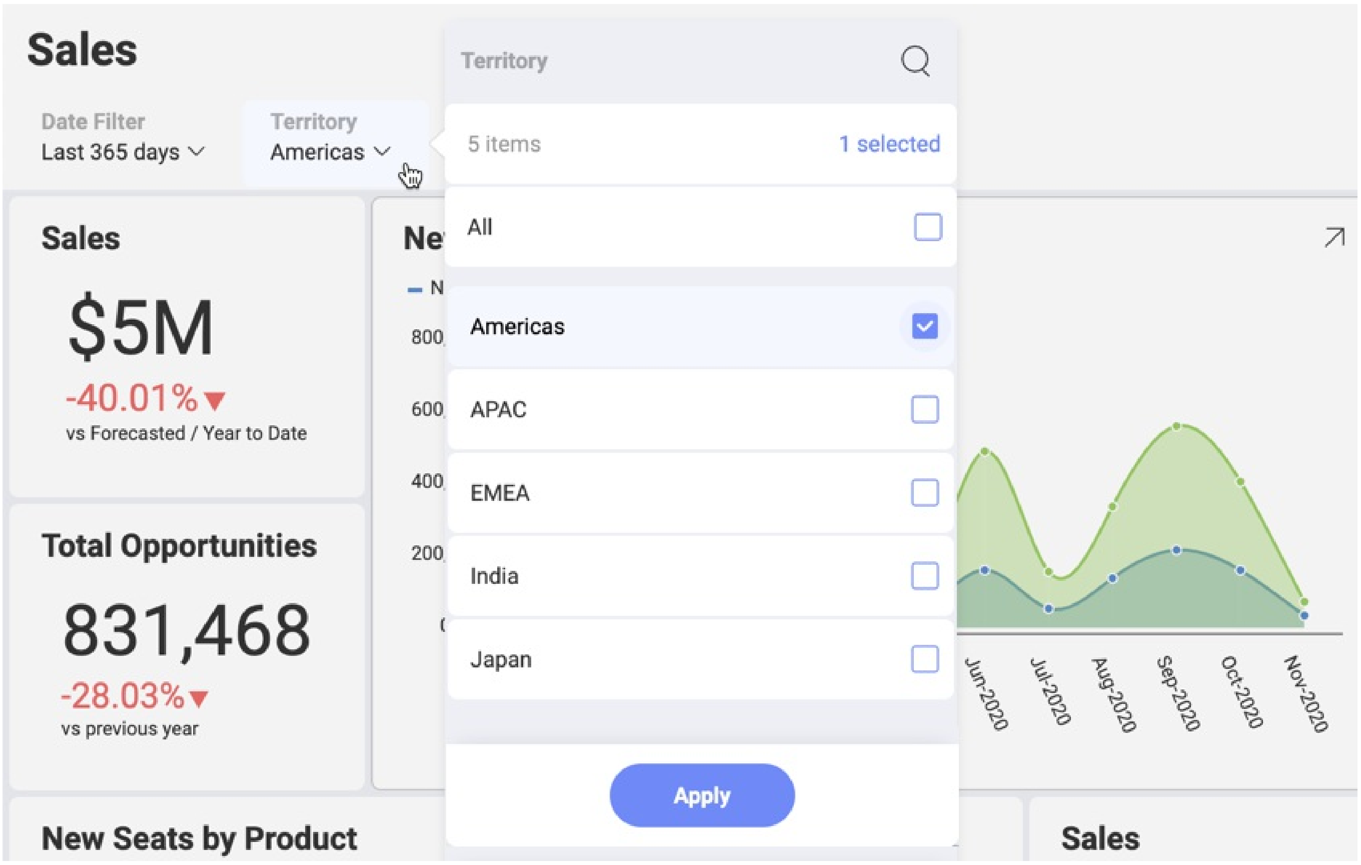
This property can be used to show or hide the Add Date Filter menu item.



```
revealView.canAddDateFilter = false;
```

Preselected Filters

You can specify which values are initially selected among existing Dashboard Filters when loading a dashboard.



The following code snippet illustrates how to load a dashboard “AppsStats”. By setting the “Territory” dashboard filter’s selected value to be “Americas”, the dashboard will be showing data filtered by “Americas”.

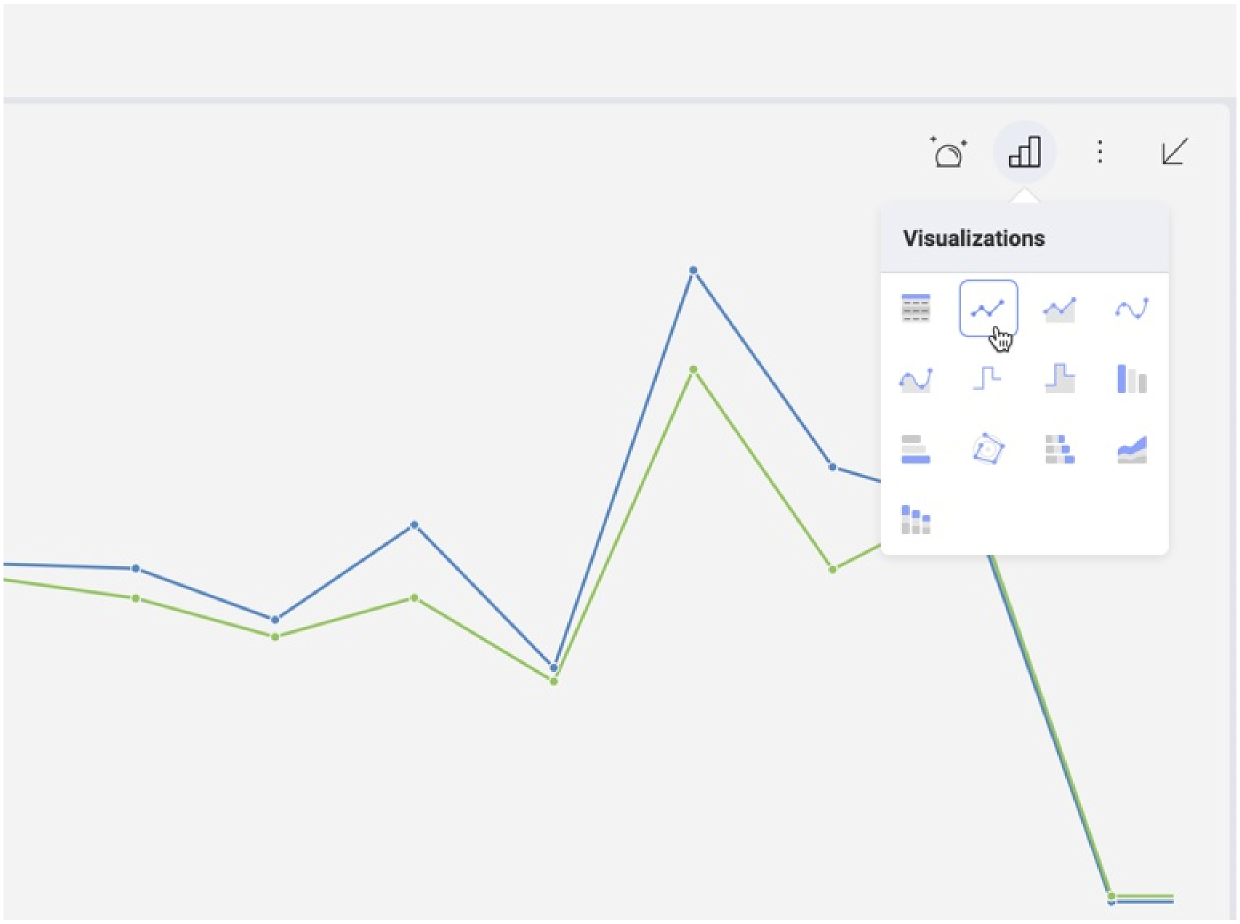
```
var dashboardId = "AppsStats";

$.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {
    dashboard.filters.getByTitle("Territory").selectedValues = ["Americas"];

    var revealView = new $.ig.RevealView("#revealView");
    revealView.dashboard = dashboard;
}, function (error) {
});
```

availableChartTypes

This property can be used to filter the visualization types available to the user.



You can, for example, add or remove visualizations as shown below:

```
revealView.availableChartTypes.add($.ig.RVChartType.bulletGraph);
revealView.availableChartTypes.remove($.ig.RVChartType.choropleth);
```

In addition, you can use a brand new Array that includes only the visualizations you want to be available:

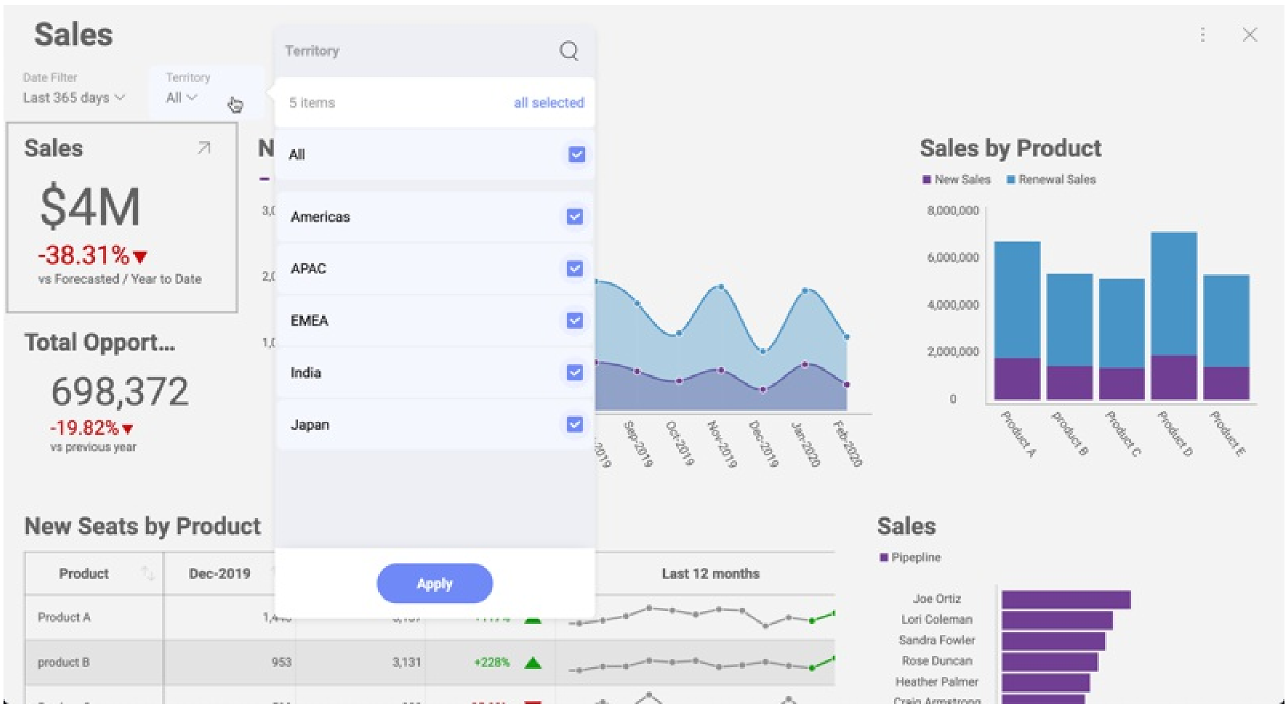
```
revealView.AvailableChartTypes = [$.ig.RVChartType.bulletGraph, $.ig.RVChartType.choropleth];
```

Setting up Dynamic Filter Selections

Overview

Sometimes your application will integrate a custom UI, to present the user with a list of values to select. And you might want that user selection to be synchronized with a filter in the dashboard.

For example, you can have a Sales dashboard that changes figures based on the current Territory and a custom UI to select the Territory. After the user selection changes, you'd want the Sales dashboard to reflect that change. Most of the times, you would hide the filter selection normally displayed in the dashboard. This way the user won't be confused with two different ways to change the Territory in the screen.



In the following code snippet, you'll find details about how to achieve the described scenario:

```
<script type="text/javascript">
   var dashboardId = 'Sales';

   $.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {
      var revealView = window.revealView = new $.ig.RevealView("#revealView");
      revealView.showFilters = false;
      revealView.dashboard = dashboard;
   }, function (error) {
      console.log(error);
   });
   function setSelectedTerritory(territory) {
      window.revealView.dashboard.filters.getByTitle('Territory').selectedValues = [territory];
   }
</script>

<section style="display:grid;grid-template-rows:30px auto;">
   <section style="display:grid;grid-template-columns:auto auto auto auto auto;">
      <button onclick="setSelectedTerritory('Americas')">Americas</button>
      <button onclick="setSelectedTerritory('APAC')">APAC</button>
      <button onclick="setSelectedTerritory('EMEA')">EMEA</button>
      <button onclick="setSelectedTerritory('India')">India</button>
      <button onclick="setSelectedTerritory('Japan')">Japan</button>
   </section>
   <div id="revealView" style="height:500px;" />
</section>
```

As shown above, five buttons were added at the top of the dashboard, one button for each of the territories in the dashboard: Americas, APAC, EMEA, India and Japan.

Working with Dynamic Lists

Territories like Americas, APAC, India, etc. do not change over time, but other lists of values might change. In this case, if a new Territory is added to the list, a new button will not be automatically added.

You can use **$.ig.RVDashboardFilter.getFilterValues** method to get the list of values for a given filter, in this case the following call will leave an array with five **$.ig.RVFilterValue** objects in _window.territories</emphasis>:

```
var filter = window.revealView.dashboard.getByTitle('Territory');
filter.getFilterValues(function (values) {
   window.territories = values;
}, function (error) {
   console.log(error);
});
```

You can then use the *label* attribute from **$.ig.RVFilterValue** to display the name of the territory and the **values** attribute to set the selection in the filter. The following code snippet shows how to populate a ComboBox to automatically select the Territory:

```
<script type="text/javascript">
    var dashboardId = 'Sales';

    $.ig.RVDashaboard.loadDashboard(dashboardId, function (dashboard) {
        var revealView = window.revealView = new $.ig.RevealView("#revealView");
        revealView.showFilters = false;
        revealView.dashboard = dashboard;

        var filter = revealView.dashboard.filters.getByTitle('Territory');
        filter.getFilterValues(function (values) {
            window.territories = values;
            var buttonsPanel = $('#buttonsPanel')[0];
            for (var i = 0; i < values.length; i++) {
                var button = $('<button onclick="setSelectedTerritory(window.territories[' + i + '].values)">' + values[i].label + '</button>');
                buttonsPanel.append(button[0]);
            }
        }, function (error) {
            console.log(error);
        });
    }, function (error) {
        console.log(error);
    });
    function setSelectedTerritory(territory) {
        var filter = window.revealView.dashboard.getByTitle('Territory');
        filter.selectedValues = [territory]);
    }
</script>

<section style="display:grid;grid-template-rows:30px auto;">
    <section style="display:grid;grid-template-columns:auto auto auto auto auto;" id="buttonsPanel">
    </section>
    <div id="revealView" style="height:500px;" />
</section>
```

As shown above, the section containing the buttons is assigned with the “buttonsPanel” id. Then, JQuery is used to dynamically create the buttons and append them to the DOM document.

The variable *window.territories* holds the list of territories, and is later used when selecting the values in the “onclick” code for each button.

Setting Up Initial Filter Selections

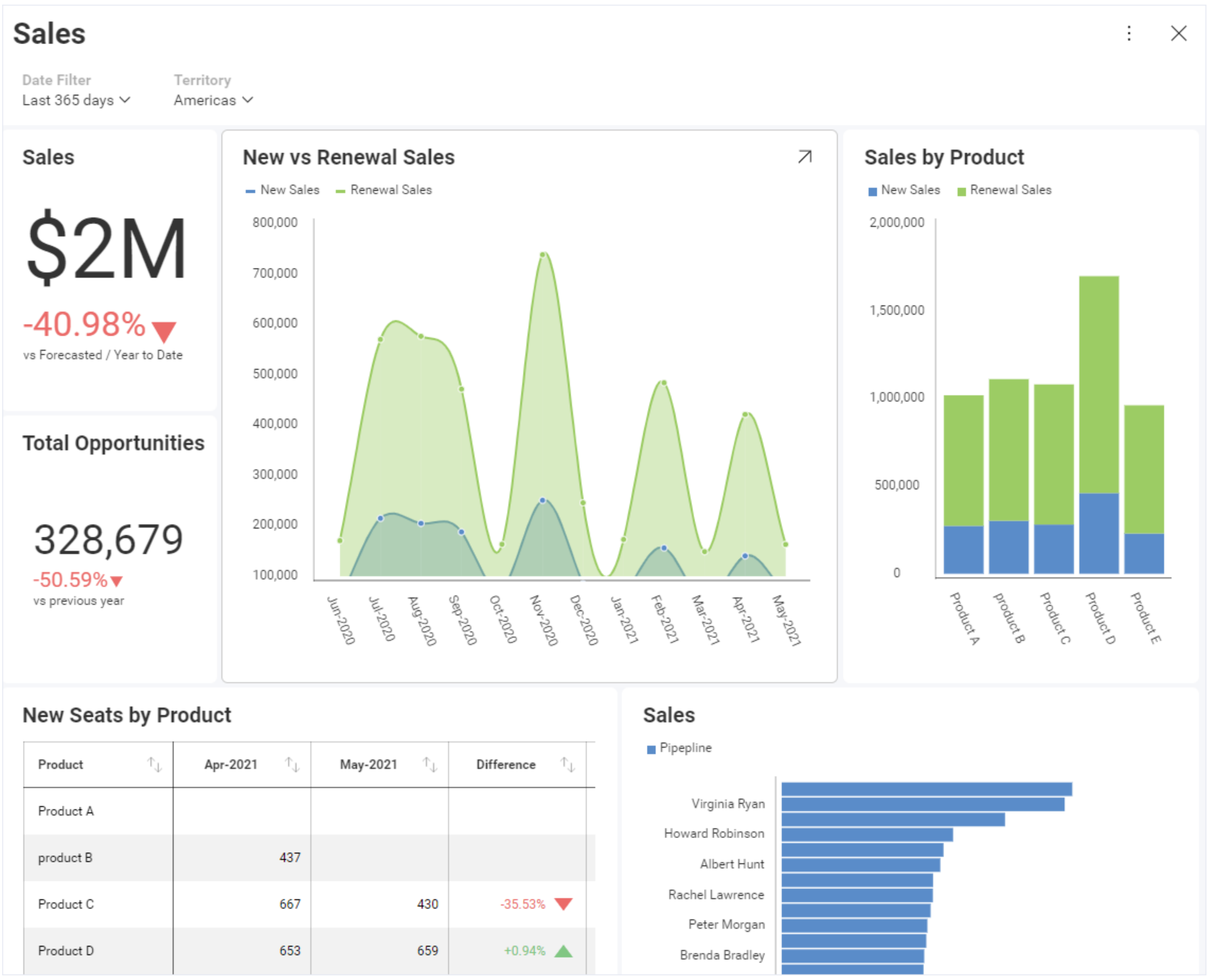
Overview

Sometimes, you want to display a dashboard with filters already applied. Dashboard filters are very useful to slice the contents of all the visualizations at once. Because of this, you can use the SDK to set up initial dashboard filter selections that remain in context for all the dashboard's visualizations.

Example Details

In this example, you have a dashboard showing Sales data with the following filters:

- A given period of time (last 365 days, Year to Date, etc.);
- Territory (Americas, Europe, Asia, etc.).



Product	Apr-2021	May-2021	Difference
Product A			
product B	437		
Product C	667	430	-35.53% ▼
Product D	653	659	+0.94% ▲

Code Example

In this case, you want to set the initial filters selection to:

- "Year to Date" (instead of "Last 365 days", the default setting for this dashboard);
- Sales associated to the Territory of the current user.

As part of the initialization process and once the dashboard is loaded, you can retrieve the list of filters in the dashboard and use these filters to set the selected values though the dashboard object and finally assign it to the revealView's dashboard property:

```
<script type="text/javascript">
   var dashboardId = 'Sales';

   $.ig.RVDashboard.loadDashboard(dashboardId, function (dashboard) {

      dashboard.filters.getByTitle("Territory").selectedValues = [getCurrentUser().territory];
      dashboard.dateFilter = new $.ig.RVDateDashboardFilter($.ig.RVDateFilterType.YearToDate);

      var revealView = new $.ig.RevealView("#revealView");
      revealView.dashboard = dashboard;

   }, function (error) {
       console.log(error);
   });
</script>

<div id="revealView" style="height:500px;" ></div>
```

Note

The code above assumes that **getCurrentUser().territory** returns the territory for the current user.
Setting the initally selected value in the date filter client-side is only supported when the dashboard **already has a date filter** created (in the .rdash file that you use).

Hiding filters

It is possible that you might not want users to access data from territories different than their own. In that case, you can restrict the access to filters by configuring the **$.ig.RevealView** object to hide the panel containing the dashboard filters:

```
revealView.showFilters = false;
```

That setting will restrict users to see data only for their associated territory.

Finally, in the case that you still want users to change the date filter selection, take a look at **Setting up Dynamic Filter Selections**. There you'll find information about how to create your own UI that, allowing the user to change the date filter.

Web Server .NET API Reference

Here you will find technical information about Reveal SDK, specifically about the Web Server .NET API. For a complete reference, please follow the link

Most commonly used classes and interfaces:

Main SDK concepts and features

RevealSdkContextBase
RevealEmbedSettings
Dashboard class

Datasources

IRVDataSourceProvider

RVSqlServerDataSource
RVSqlServerDataSourceItem

RVRESTDataSource
RVRESTDataSourceItem

RVJsonDataSource
RVJsonDataSourceItem

IRVDataProvider
RVInMemoryData
RVInMemoryDataSource
RVInMemoryDataSourceItem

Authentication

IRVAuthenticationProvider
IRVDataSourceCredential
RVBearerTokenDataSourceCredential
RVUsernamePasswordDataSourceCredential

Web Client JS API Reference

Here you will find technical information about Reveal SDK, specifically about the Web Client JavaScript API. For the complete reference, please follow the link

Most commonly used classes and interfaces

Main SDK concepts and features

RevealView

RVDashboard

RevealSdkSettings

RVVisualization

RevealTheme

RevealUtility

Datasources

RVSqlServerDataSource

RVSqlServerDataSourceItem

RVRESTDataSource

RVRESTDataSourceItem

RVJsonDataSource

RVJsonDataSourceItem

RVInMemoryDataSource

RVInMemoryDataSourceItem

Filtering

RVDateDashboardFilter

RVDashboardFilter

RVFilterValue