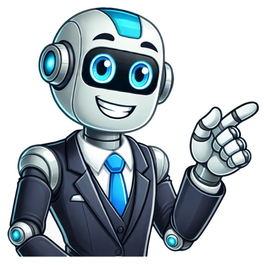
Click Here



viewer initialization and event handling ViewerJS instance must be initialized before usage. The viewerInitialized event is raised when initialization is complete. This event is recommended for adding custom event triggers. Custom JavaScript events are exposed by the Viewer component, each having a detail property. events: - pageRendered: pageNum, doc - allVisiblePagesRendered: doc - allPagesInitialized: doc - loadDocumentError: reason, url, file, evt document - documentReleased: - pagePrinted: evt.detail pagePrinted: evt.detail viewerinitialized Retrieving the web viewer JS instance by ID const viewerId = 'yourViewerId'; const viewer = window.TWPdfViewerUtil.getViewer(viewerId); if (!viewer) { console.log(`viewer with id ${viewerId} not found.`); } else { console.log(`viewer with id ${viewerId} found.`); } Note: viewer instance is ready for safe usage only after event twWiewerInitialized. ###ARTICLEthis to false viewer.toolbar.isBookmarksVisible = false; Document info properties const viewer = window.TWPdfViewerUtil.getViewer('yourViewerId'); viewer.showDocumentProperties(); // ... viewer.closeDocumentProperties(); // if you want to hide to the visitor show document info button set this to false viewer.toolbar.isDocumentInfoVisible = false; NOTE In current version isn't possible to get document properties object directly from JS. But, if your use-case scenario needs it, please contact us with the details about your use case and we will try to add it. Page rotation const viewer = window.TWPdfViewerUtil.getViewer('yourViewerId'); viewer.rotateClockwise(); // rotate for 90 degrees viewer.rotateCounterClockwise(); // rotate for -90 degrees // you can hide rotation buttons from the toolbar viewer.toolbar.isRotateClockwiseVisible = false; viewer.toolbar.isRotateCounterClockwiseVisible = false; Single page VS Multi page view const viewer = window.TWPdfViewerUtil.getViewer('yourViewerId'); viewer.activateSinglePageView(); // ... viewer.activateMultiPageView(); // you can hide view buttons from the toolbar viewer.toolbar.isSingleViewVisible = false; viewer.toolbar.isMultiViewVisible = false; Zoom const viewer = window.TWPdfViewerUtil.getViewer('yourViewerId'); //available zoom levels for the loaded document are based //on the size of the PDF document console.log(viewer.zoomService.zoomLevels); //max zoom value for the loaded document console.log(viewer.zoomService.getMaxZoomValue()); //increases zoom level viewer.zoomService.zoomIn(); //decreases zoom level viewer.zoomService.zoomOut(); //set zoom value to specific value viewer.zoomService.setZoomValue(65); console.log('Current scale', viewer.zoomService.currentScale); // you can hide zoom functionality from the toolbar viewer.toolbar.isZoomInVisible = false; viewer.toolbar.isZoomOutVisible = false; viewer.toolbar.isZoomDropDownVisible = false; Page text extraction and text selection const viewer = window.TWPdfViewerUtil.getViewer('yourViewerId'); //disable text selection viewer.disableSelection(true); // extract all text from the 1st page const pageText = viewer.getPageTextContent(1); NOTE In current version isn't possible to get already selected text or to select some text directly from JS. But, if your use-case scenario needs it, please contact us with the details about your use case and we will try to add it. Retrieve viewer by ID const viewer = window.TWPdfViewerUtil.getViewer(viewerId); if (!viewer) { console.log(`viewer with id ${viewerId} not found.`); } else { console.log(`viewer with id ${viewerId} found.`); } Retrieve all viewers const viewers = window.TWPdfViewerUtil.getActiveViewers(); for (let i = 0; i < viewers.length; i++) { console.log(`viewer with id ${viewers[i].id}`); } A few load document event examples const viewer = window.TWPdfViewerUtil.getViewer(viewerId); viewer.viewerDiv.addEventListener("pageRendered", function (evt) { console.log(`page number:${evt.detail.pageNum} for document ${evt.detail.doc} has been rendered.`); }); // optional custom load document events viewer.viewerDiv.addEventListener("allVisiblePagesRendered", function (evt) { console.log("allVisiblePagesRendered", evt.detail.doc); }); viewer.viewerDiv.addEventListener("allPagesInitialized", function (evt) { console.log("allPagesInitialized", evt.detail.doc); }); viewer.viewerDiv.addEventListener("loadDocumentError", function (evt) { console.log("loadDocumentError", evt.detail.url, evt.detail.reason, evt.detail.file, evt.detail.evt); }); viewer.loadDocumentFromUrl("test-pdf/pdfprint-manual.pdf"); // alternatives for load: // viewer.loadDocumentFromFile(file, password, viewer); // viewer.openThroughDialog(); // if you want to hide open document button from the toolbar viewer.toolbar.isOpenVisible = false; PDF is a very old file format that has been used for years to draft information in almost all sectors, whether healthcare, legal, education, or entertainment. This blog introduces how a developer can create a PDF file from scratch using C# and later display and edit the PDF file on the client side using a Javascript-based viewer. Grapecity Documents provides document solutions for various formats such as PDF, Word, Excel, and Imaging, which lets you either create these files through code or edit the existing ones as per your requirements. Hence, these APIs can be used to automate the document workflow of creating, modifying, saving, and viewing documents of different types. In this blog, we will use GrapeCity Documents for PDF to create a PDF file and GrapeCity Documents PDF Viewer to load, view, and edit the PDF file on the client side. So, let’s just take each of these tasks one by one. Ready to Get Started? Download GCDocuments Today! GrapeCity Documents for PDF, referred to as GcPdf hereafter, offers a powerful API that lets you create or edit a PDF file, offering many popular features such as Annotation, Forms, Digital Signatures, Layers, etc. You may refer to the documentation and demos for detailed information about all the features offered by GcPdf API. The steps below will help you understand the basics of creating and saving a PDF file using GcPdf API. Open Visual Studio 2022 and create a new ASP. NET Core Web Application, choosing the appropriate project template and providing an appropriate project name, here it is CreatePDF. Choose .NET Core 6.0 as the project's target framework. As discussed above, we will be using GrapeCity Documents for PDF to create a PDF file on the server side. So, we must install the Grapecity.Documents. PDF package to work with the PDF API. To accomplish the same, right-click the project in Solution Explorer and choose Manage NuGet Packages. In the Package source on the top right, select NuGet Gallery | Home. Click the Browse tab on the top left and search for "GrapeCity/Documents", now select GrapeCity.Documents.Pdf from the left panel. Install it by clicking on the Install button in the right panel. Once the package gets installed, we will move on to defining the code used to generate the PDF file. Open Index.cshtml.cs page found under the Pages folder in the project folder. We will be defining the server-side code in this file to generate the PDF file. Add the following code to initialize the Environment variable of type IWebHostEnvironmet to save the generated PDF file in the web root folder. This is done so as to make sure that the client-side viewer can easily access the PDF file, as explained in the next section of the blog. //Define Environment variable to access web root folder private IWebHostEnvironment Environment; public IndexModel(ILogger logger, IWebHostEnvironment _environment) { _logger = logger; Environment = _environment; CreatePDF(); } ```we define a method to create a PDF file. The code snippet below defines this method and creates a single-page PDF file by adding a background image and a bullet list over the image. You can find detailed information about all the API members used to accomplish the task via the code comments added to the code snippet below: public void CreatePDF() { const int FontSize = 12; //Define an instance of GcPdfDocument var doc = new GcPdfDocument(); //Add a new page var page = doc.Pages.Add(); var g = page.Graphics; //Initialize TextLayout to render text var tl = g.CreateTextLayout(); //Add an image to PDF document var img = Image.FromFile(Path.Combine("Resources", "ImagesBis", "2020-website-gcdocs-headers_tall.png")); var rc = page.Bounds; rc.Height *= 0.65f; g.DrawImage(img, rc, null, ImageAlign.StretchImage); //Define text format settings var ip = new PointF(48, 72); var font = Font.FromFile(Path.Combine("Resources", "Fonts", "OpenSans-Regular.ttf")); var tfCap = new TextFormat() { Font = font, FontSize = FontSize * 1.6f, ForeColor = Color.White }; var tf = new TextFormat() { Font = font, FontSize = FontSize, ForeColor = Color.White }; tl.MaxWidth = 72 * 5; // Add Header: tl.AppendLine("Fast, Efficient Document APIs for .NET 5 and Java Applications", tfCap); tl.AppendLine(tfCap); tl.AppendLine("Take total control of your documents with ultra-fast, low-footprint APIs for enterprise apps.", tf); tl.AppendLine(tf); g.DrawTextLayout(tl, ip); // Add Bullet list: ip.Y += tl.ContentHeight; tl.Clear(); const string bullet = "\x2022\x2003"; tl.FirstLineIndent = -g.MeasureString(bullet, tf).Width; tl.ParagraphSpacing += 4; tl.Append(bullet, tf); tl.AppendLine("Generate, load, edit, save XLSX spreadsheets, PDF, Images, and DOCX files using C# .NET, VB.NET, or Java", tf); tl.Append(bullet, tf); tl.AppendLine("View, edit, print, fill and submit documents in JavaScript PDF Viewer and PDF Editor.", tf); tl.Append(bullet, tf); tl.AppendLine("Compatible on Windows, macOS, and Linux", tf); tl.Append(bullet, tf); tl.AppendLine("No dependencies on Excel, Word, or Acrobat", tf); tl.Append(bullet, tf); tl.AppendLine("Deploy to a variety of cloud-based services, including Azure, AWS, and AWS Lambda", tf); tl.Append(bullet, tf); tl.AppendLine("Product available individually or as a bundle", tf); //Render text g.DrawTextLayout(tl, ip); //Save the document to web root folder doc.Save(Path.Combine(Environment.WebRootPath, "sample.pdf")); } The screenshot below depicts the PDF file generated by executing the above code, loaded in Adobe Acrobat Reader: Load and View PDF in GcPdfViewer Next, we move on to loading and viewing this PDF file generated on the server side into a Javascript-based viewer, i.e., GrapeCity Documents PDF Viewer, referred to as GcPdfViewer, hereafter. GcPdfViewer is a Javascript based viewer used to load, view, and edit PDF files on the client side. It offers a powerful API to support all the edit operations on the client side. Refer here for more details on the API members. The steps below will guide you on how to install and use GcPdfViewer. We begin with installing GcPdfViewer in our project to start working with the respective JS files for GcPdfViewer. To accomplish the same, open Package Manager Console, choosing Tools → NuGet Package Manager → Package Manager Console. Run the following command to install GcDocs PDF Viewer. Ensure that the directory location in the command prompt is set to the lib folder in the project. npm install @grapecity/gcpdfviewer The GcDocs PDF Viewer will be installed in _\\wwwroot\libode_modules_ folder. In this step, we add the following code to the Index.cshtml file by replacing any existing code. The code below adds the GcPdfViewer JS file to access all the client-side APIs to initialize and work with the viewer. Next, we create an instance of GcPdfViewer by initializing the GcPdfViewer class and passing the DIV element id, which is to be used to render the viewer. Later, we add the default panels using the addDefaultPanels method and load the PDF file by invoking the open method of the GcPdfViewer class. Please note that the open method looks for the PDF file in the web root folder, i.e., why we saved the server-side generated PDF file in the wwwroot folder. window.onload = function () { var viewer = new GcPdfViewer("#root", { /* Specify options here */ }); viewer.addDefaultPanels(); viewer.open("sample.pdf"); } The screenshot below depicts the PDF file loaded into Javascript-based GcPdfViewer after executing the above code snippet: Now that we have loaded the PDF file into the Javascript-based GcPdfViewer let’s explore the editing possibilities offered by the viewer. GcPdfViewer, by default, only supports viewing the PDF file in the viewer. To enable the editing tools, we must configure the SupportAPI for the viewer. SupportAPI is a server-side library that, when connected to GcPdfViewer, lets you edit the PDF file on the client side using all the editing tools available in the viewer. Refer to the following documentation topic, which helps you understand how to configure the PDF Editor. GcPdfViewer provides many editing tools such as annotation editor, collaboration, form editor, signature, etc. Refer to the documentation and demos for exploring all the available editing tools. Here, we quickly have a look at two common ways to edit the PDF using GcPdfViewer editing tools. Add Annotations using Annotation Editor: The use of annotations in PDF is a very popular way of editing the PDF file as it helps in many scenarios, such as reviewing a PDF file, sharing additional information through comments or sticky notes, and highlighting different parts of the text to emphasize specific content in the document. GcPdfViewer provides an annotation editor to add or remove different types of annotation to the document, such as text annotation, circle annotation, stamp annotation, redact annotation, etc. You can refer to the following link to find details about all the available annotations. Here in the GIF below, we make use of the circle annotation to highlight an issue we observe with the document: Organize PDF Pages using the PDF Organizer Tool Another common editing requirement when working with PDF files is to add or delete pages or may be merge two PDF files, or delete specific page ranges. GcPdfViewer supports page organization using the Page tool available in the toolbar, which, when clicked, opens a secondary toolbar with different page editing features such as adding new documents/pages, deleting the current page, and a page organizer tool to organize pages likeTo create a secure document viewer in ASP.NET Core without showing the physical path, you can use the DocumentUltimate library. Here's an example: To enable PDF rendering using the DocumentViewer └ TX Text Control .NET Server for ASP.NET └ Web.MVC.DocumentViewer Namespace └ DocumentViewer Class in combination with PDF.js to render PDF documents, a new ASP.NET Core Web App must be created. The latest version of Visual Studio 2022 that comes with the .NET 6 SDK or .NET 7 SDK should be used for this purpose.