PDFTRON

WebViewer Developer Guide

version 2.2.0

PDFTRON

Table of Contents

1. Introduction

Audience and Scope

This document is intended for developers wishing to integrate PDFTron WebViewer with their Web application. It covers the basic usage of the WebViewer.js wrapper, as well as discusses advanced customizations to the Desktop and Mobile viewers.

WebViewer Components

PDFTron WebViewer is a cross-platform solution that offers a seamless and effective way to embed viewing of PDF and other document formats directly within your Web application. There are three major components to this solution:

1. The client-side viewer application
2. The document conversion component (if necessary)
3. Your own server for document hosting and annotation handling (optional)

Client-side Viewer Application

The PDFTron WebViewer is a solution that delivers consistent and reliable document viewing across modern browsers. WebViewer uses HTML5 technology to take advantage of the latest browser features. For pure document viewing, the WebViewer does not require any server-side scripting. The document files can be served directly from any static Web server.

Platforms Supported

- HTML5 (any modern browser with Canvas support; e.g. IE9+, Chrome, FireFox, Safari, Opera).
- Web browsers on iOS (iPad/iPhone), Android and other mobile platforms.
- Native mobile SDKs on Android and iOS.
- .NET/WPF, C/C++, JAVA apps on Windows, Linux, and Mac OS X.

WebViewer Document Backends

WebViewer supports two document backends. It can view PDF documents directly using the PDFNetJS backend and it can support viewing a wide range of document file formats (including PDF) by converting all documents to the XOD format. There are benefits to both approaches in different situations which are outlined below.

PDFNetJS

Using PDFNetJS as the document backend allows you to leverage the power of PDFNet rendering in the browser. When using this backend there is no conversion of documents required. The trade off is that only PDF documents can be viewed and PDFNetJS is not currently supported on mobile devices.

Viewing PDF files with PDFNetJS is licensed differently from the XOD backend and requires a license key to be specified when creating the viewer. Without a license key a demo stamp is applied.

XOD Backend

Using the XOD backend allows many different file formats to be viewed as long as they are converted to the XOD format. The XOD conversion step performs optimizations on the file which allows fast and accurate rendering on both desktop and mobile devices.

PDFTron provides the XOD conversion component in three forms. You can choose the one that suits your deployment, requirements and licensing needs.

PWS Cloud

PWS Cloud is a RESTful web service that allows quick and simple XOD conversions through a REST API. You get the added benefit of having the XOD documents hosted for you on the PWS Cloud server, along with a hosted version of WebViewer. This is the easiest and most cost-efficient way of converting documents.

With PWS Cloud, you can create XOD documents by making HTTP POST requests to the server. For sample code snippets on how to do this, please see the following link:
http://www.pdftron.com/pws/cloud/snippets.html

For more information, see http://www.pdftron.com/pws/cloud.
[Licensing: Available with a PWS subscription]

PDFNet SDK with WebViewer Add-on

With this option, you get the benefit of using PDFNet SDK, our full-featured PDF SDK, on your server. Essentially you would be doing your own XOD conversions on your server with PDFNet. You can do more advanced things here, pre-processing the document: e.g. apply your own watermark to all converted documents, add/remove pages to converted documents, merge annotations back to the original PDF. This gives you the most flexibility and control over your documents.

You can convert your documents to XOD documents using the pdftron.PDF.Convert class. Specifically, the method Convert.ToXod() converts the input file to XOD format and saves to the specified path.

To specify the convert options for the conversion, the user should use the Convert.XODOutputOptions class and pass it as a parameter in the ToXod() method.

For more information, see http://www.pdftron.com/pdfnet
[Licensing: Available with Redistributable Licensing or a PWS Subscription (On-Premises)]

DocPub Command-line Converter

With this option, you run the conversions with a command-line tool. This works best if you have a static set of documents you want to convert in a batch, and upload to a server all at once.

For more information, see http://www.pdftron.com/docpub/index.html
[Licensing: Available with Redistributable Licensing or a PWS Subscription (On-Premises)]

PWS On-Premises

PWS On-Premises is an alternative licensing model for PDFNet SDK and DocPub CLI. You can take advantage of the pay-as-you-go model that a PWS subscription offers while keeping your documents secure by performing conversions in-house.

For more information, see http://www.pdftron.com/pws/onpremise.html.

Table 2. XOD Conversion Features by Converter Types

	PDFNet SDK	DocPub CLI	PWS Cloud
XOD Conversion	✓	✓	✓
XOD Encryption	✓	✓	✓
MS Office Documents Support	✓ (Windows only)	✓ (Windows only)	✓
Hosted Viewer			✓
Hosted XOD			✓
Document Pre-Processing (PDF)	✓		
Built-in Annotation (XFDF) merging	✓		
On-the-fly Conversion Streaming	✓	✓	

Your Server Component

In order to view your documents on the web, both the viewer application (WebViewer) and your PDF or converted XOD documents need to be hosted on a web server. Moreover, if you need access control over your documents or manage user-created annotations, you will need your own server to handle this logic.

WebViewer and Document Hosting

With PDFNet or DocPub, you will be hosting your own version of WebViewer and documents on your own web server. If you choose PWS Cloud as your document conversion component, you do not have to host your own viewer application or XOD documents as they are all hosted by PWS Cloud. (You could also download converted XOD documents from PWS Cloud and host it yourself)

There are a couple of things to keep in mind when hosting your own WebViewer and XOD documents. For the best performance, ensure that your web server supports the Range request header. WebViewer makes byte range requests to download only parts of the XOD file at a time; this allows the WebViewer to start rendering documents without having to download the entire file first. Also, make note that ideally the WebViewer application and the XOD files should be hosted in the same domain host to avoid the same origin policy restrictions of the browser.

For WebViewer with PDFNetJS the entire PDF file is downloaded up front so support for range requests is not necessary.

Annotation Handling

A big area of interest for the WebViewer is support for annotations. WebViewer provides a fully-functional annotation framework in HTML5 that is compatible with PDF XFDF annotations.

While the WebViewer is provided as a pure client application, it also has built-in support for communicating with a back-end server. If an annotation server path is specified in the WebViewer, it will issue AJAX requests to fetch and save the annotations.

Since the server-side handling of annotations is heavily linked with the application logic, it is up to the developers to implement. WebViewer provides a sample implementation of a PHP annotation handler. Please refer to the section on Annotations in this document for more details.

2. WebViewer Quick Start

At the heart of it, WebViewer aims to provide document viewing capabilities in modern desktop and mobile devices.

Under the hood, DocumentViewer is the core API that renders the document and creates canvas elements for each page. DocumentViewer by itself is powerful and can be used to implement custom viewing experiences. (See the WebViewer samples “Flipbook Demo”)

ReaderControl on the other hand, is a full-featured UI control that builds on top of the DocumentViewer. ReaderControl adds common interface elements expected of a document reader: a tool bar for navigation and panels for thumbnail views, bookmarks and annotations.

WebViewer.js is an API wrapper that allows any ReaderControl type to be created and controlled via a single interface. It greatly simplifies the creation of the viewer and also provides utilities like the detection of the browser support.

For the best user experience out-of-the-box, it is recommended to use WebViewer.js. However, each of the components can be used individually to suit your use case.

Using JavaScript with WebViewer

To create your own WebViewer web page from scratch follow these steps.

1. Create an HTML page.

2. Add the necessary scripts to the <head> tag of the HTML page.

```
<script src="jquery-1.7.2.min.js"></script>
<script src="WebViewer.min.js"></script>
```

Alternatively you can include the following un-minified version of WebViewer.js

```
<script src="WebViewer.js"></script>
```

WebViewer.js depends on jQuery, so it must be included. Including WebViewer.js (or WebViewer.min.js) will allow you to use the WebViewer class in your JavaScript code.

3. Create a tag in the HTML <body> and give it an id. This will be the container for the web viewer.

```
<div id="viewer"></div>
```

4. Add the following script to create a new instance of the WebViewer.

```
<script>
      $(function() {
          var viewerElement = document.getElementById('viewer');
          var myWebViewer = new PDFTron.WebViewer({
              type: "html5",
              initialDoc: "GettingStarted.xod"
            }, viewerElement);
      });
</script>
```

The script above will create and render a PDFTron WebViewer control on the HTML page as a child of the provided element. The viewer will then load the document as specified by initialDoc immediately.

As a start, you can load the sample file “GettingStarted.xod” included in the download package.

To load a PDF file just set initialDoc to a pdf document (e.g. “GettingStarted.pdf”). To display PDF documents without a demo stamp you’ll need to input your PDFNetJS license key as an option, e.g. `l: "Your License Key Here"`

5. It is now possible to use the “myWebViewer” variable to invoke ReaderControl’s methods. For example, the following code demonstrates how to load another document:

```
myWebViewer.loadDocument("myOtherFile.xod");
```

Additionally, you can use jQuery event handlers to react to changed events.
For example, the following code shows how to be notified when the page changes.

```
$(viewerElement).on('pageChanged', function(event) {
    alert("Current page is: " + myWebViewer.getCurrentPageNumber());
});
```

With the WebViewer methods and event binding, you have the power to create your own GUI interface*.
Please refer to the API documentation for other methods and events which can be used. The API reference can be found under doc/ or online at:
http://www.pdftron.com/webviewer/demo/documentation.html

*Please note that the HTML5 Mobile viewer is optimized for mobile devices and therefore cannot be controlled at as fine a level through WebViewer.js. If WebViewer.js detects that the user agent is a mobile device, it will automatically switch to the Mobile viewer.

6. Save the HTML page under your web server. Make sure that the page is running on your web server (through HTTP or HTTPS).

3. WebViewer Features

Because the viewer is built with pure HTML, CSS and JavaScript, it is very customizable. There is no compilation involved: simply specify custom JavaScript files.

Customization Framework

WebViewer was designed for easy customization of the user interface and custom features. The ReaderControl allows you to specify a configuration file to run any custom JavaScript.

Configuration Options

For common user interface customizations, WebViewer makes this easy by loading configuration options before initializing ReaderControl. Some configurable options include hiding the toolbar, hiding the side panel, providing a server URL for annotation saving, and alert message strings.
To find out more about the options available, please see “ReaderControl.config” in the WebViewer HTML5 API Reference. By default, the viewer will read the options in ReaderControlConfig.js. You can either change this directly, or load an external configuration file.

External Configuration File

To make customizations simple and clean, you can define all your customizations in an external JavaScript configuration file. For example to instantiate the viewer with a config file:

```
var myWebViewer = new PDFTron.WebViewer({
    config: “path/to/my/config/file.js”,
    …
}, viewerElement);
```

This is the preferred way to make customizations. In fact, all the WebViewer HTML5 samples use this method of customization. The samples show a wide range of customizations, from creating annotations on the fly to adding custom buttons and other functionality.

Pass Custom Data through WebViewer

To pass custom data to the HTML5 viewers you can add a “custom” property to the WebViewer options that are passed to the constructor. For example to pass a custom object add this property:

```
custom: JSON.stringify({data: 10})
```

Then in a config file you can add the following line to retrieve it:

```
var custom = JSON.parse(window.ControlUtils.getCustomData());
```

Customization with PWS Cloud

If you choose to use PWS Cloud for your XOD conversions, your documents with be loaded with a hosted WebViewer instance. While you cannot modify this WebViewer instance directly, you can use the external configuration file setting to make your customizations. To do so, log in to the PWS Cloud

management console and set a URL path for your viewer configuration.

Annotations and Forms

PDFTron WebViewer offers a powerful annotations framework that enables your application to have interactive documents. The WebViewer annotation framework includes features for both markup annotations and form widgets.

Markup Annotations

With the PDFTron WebViewer, users can annotate their documents freely. The annotations can be saved to an XFDF (XML Forms Data Format) file from the WebViewer, and be loaded back into the document the next time it is opened. The WebViewer runs in a client-server architecture. Some common use cases include:

- creating and saving annotations for a document
- loading annotation files (XFDF files) into a document
- collaboration: multiple users can view the same document, adding their own annotations, while seeing the annotations that others have added
- merging annotations from an XFDF file into a document
- merging annotations from different XFDF files into one single XFDF annotations file

Permission Checking

When the WebViewer first loads a document, some user information is passed to it through the URL. This includes the “user” and “admin” attributes. “user” specifies the user name of the current user viewing the document, while “admin” specifies whether the user has administrative privileges.

The WebViewer has two levels of user permissions: admin and normal. Users with admin level rights can do anything with annotations with no restrictions. Normal users on the other hand, are restricted to editing the annotations that they put into the document. They are not allowed to modify or delete annotations created by other users. Alerts will show up when they try to perform illegal operations on the WebViewer. A special case is when the author of an annotation is undefined or null. In this case, every user has permission to edit the annotation.

Read-only Mode

Read-only mode can be enabled by adding ‘enableReadOnlyMode: true’ as a property on the options object passed to WebViewer. In read-only mode, existing annotations on the document cannot be deleted or modified in any way, regardless of the permission level of the current user. However, the user is still able to select them and read their notes if they have any. Furthermore, new annotations cannot be added into the document. Read-only mode ensures that existing markups are not changed

and remain the only annotations on the document.

Note that the annotations toggle button can still be used in read-only mode to toggle all the annotations on or off.

Toggle Annotations

With the toggle annotations button located on the notes panel, the user can toggle between showing and hiding all the annotations on the currently displayed document. The keyboard shortcut to do this is Alt + T.

By default, annotations are toggled on. Depending on the current state, the toggle annotations button will show either 'Show' or 'Hide'. Note that when annotations are toggled off, new annotations cannot be added to the document unless visibility is enabled again. This is to ensure that no annotations are displayed when the user has opted to hide all annotations. Also note that while annotations do not appear on the document when annotations are toggled off, they are not removed from the document; they are merely visually removed from the viewer.

Forms

The PDFTron WebViewer provides support for interactive forms, sometimes known as AcroForms. An AcroForm is simply a collection of fields for gathering information interactively from the user. In a PDF document there may be any number of fields appearing on any combination of pages. The combined fields make up a single interactive form that can be imported or exported from the document.

During the conversion process from PDF to XOD, the form fields' name-value pairs, as well as all the information needed to recreate the fields' appearances, are saved into the internal XFDF embedded in the XOD document. This information that is stored inside the internal XFDF is used by the WebViewer to recreate the field widget elements on the viewer.

Here are the major features of the PDFTron WebViewer form support:

- rendering of the form field widgets as from the original PDF document
- dynamic data entry into form field widgets
- loading and saving of form field data
- support for a number of form actions
- programmatic access to form field data, values and child widgets via the Annotations.Forms.FieldManager class

Supported Field Widget Types

The PDFTron WebViewer supports all the form field types outlined in the PDF specification, except the signature field.

Button Fields

A button field is an interactive control that the user can manipulate with the mouse. They include the following:

- Push button: a purely interactive control that responds to user inputs without retaining a permanent field value
- Checkbox: a control that can be toggled between two states: on and off
- Radio buttons: a group of related toggles. Selecting any one from the group automatically deselects all the others, such that at most one may be on at any given time

Text Fields

A text field is a box or space in which the user can enter text by using the keyboard.

Choice Fields

A choice field contains one or more text items, where at most one of which may be selected as the field value. They include the following:

- list box: a scrollable control listing all the items that can be chosen
- combo box: a dropdown menu containing all the items that can be chosen

Supported Form Actions

The PDFTron WebViewer supports a subset of the standard PDF form action types that can be attached to form widgets. These include the following:

Submit Form Actions

A submit form action transmits the name-value pairs of selected interactive form fields to a specified URL, presumably the address of a server that will process the submitted data and send back a response. The form data may be submitted in either HTML Form format, or XFDF format. If the export type is specified as FDF format, the WebViewer would default the export format to HTML Form format.

Reset Form Actions

A reset form action resets selected interactive form fields to their default values.

JavaScript Actions

A JavaScript action causes a script to be executed when the widget is clicked. The script can be any JavaScript that is stored in the action attribute of the widget or field in the PDF document, and almost all available triggers (document open, page change, field value change, click, keypress, mouseover…) are supported. When such a trigger is fired, the JavaScript will execute in a separate global scope that simulates a subset of the PDF JavaScript API, allowing for complex interactive documents to be viewed.

Hide Actions

A hide action either hides or shows a widget element on the screen.

GoTo Actions

A GoTo action jumps the viewer to a specific page.

URI Actions

A URI action opens a URI in a separate browser window.

Document Named Actions

A document named action allows the viewer to go to a certain page. The WebViewer supports NextPage, PrevPage, FirstPage and LastPage.

If using XOD files note that these form actions must be attached to the field widgets before the document is converted from PDF to XOD. The attributes describing the action are exported into the widget XFDF elements, and the WebViewer will create appropriate event handlers upon reading these action attributes while loading the XFDF file.

Processing Annotations and Forms

Once users create markup annotations and fill in form fields, you will want to manage and process this data. In the sections below we discuss how to save, load and merge annotation data in WebViewer.

Loading Annotations and Form Fields

If using XOD files then during the conversion process, an XFDF file is embedded into the XOD document (internal XFDF), which stores all the existing annotations, links, and form data of the PDF document.

When a XOD document is first loaded into WebViewer, it looks into the internal XFDF embedded into the XOD document itself during the convert process, and uses that XFDF to load all of the following stored in the XFDF: annotations, links, and form field widgets. Please note that ‘enableAnnotations: true’ must be set as a property on the options object passed to WebViewer so that both annotations and widgets are loaded in the viewer. On the other hand, links are loaded in the viewer regardless of the value of ‘enableAnnotations’.

While the user can provide an external XFDF file to load annotations from the onDocumentLoaded() callback function in ReaderControl.js, it is important to note that this external XFDF file would replace the internal XFDF file as the source of annotations and form data loading. That is, only the annotations and widgets stored in the external XFDF would get loaded, while the internal XFDF would be ignored. Therefore, if the original PDF document contains an AcroForm, the external XFDF the user provides must contain the form field widgets information inside it as well so that the PDFTron WebViewer can recreate the form field widgets.

Exporting Annotations and Form Fields

WebViewer allows multiple methods of exporting annotations and form fields in order to accommodate the different needs of users. Both annotations and form field data are exported into one single XFDF file. Here are the 3 most common methods to export annotations for XOD documents. Note that when using WebViewer with PDFNetJS you can also easily download the current PDF document with annotations.

1. Export to XFDF as local download

The user can download the XFDF file containing the annotations and form data of the document directly from the WebViewer, by the use of dataURLs.

2. Export whole XFDF to server

The user can export the whole XFDF file (as a string) to the server, where this copy would replace the central copy stored in the server for the document.

3. Export modified XFDF to server

WebViewer supports the export of only the modified annotations (newly added, modified existing, deleted existing) to the server as an XML command. The command would contain only the annotations and form fields that are changed. For added and modified annotations, the command would include the XFDF representation of the annotation, while for deleted annotations, the command would only include the ID of the annotation. Note that unlike the first two methods, this method does not export the information needed to recreate the form field widget appearance. Only the form field data, that is, name-value pairs, would be exported. If the user is exporting with the command structure, it is important for the server to implement some kind of XFDF merging logic so that the central XFDF copy can be updated properly.

This is the command structure used by the WebViewer:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xfdf xmlns="http://ns.adobe.com/xfdf" xml:space="preserve">
<fields />
<add />
<modify />
<delete />
</xfdf>
```

The fields element contains the modified form fields. The add, modify, delete elements contain the added, modified, and deleted annotations respectively.

Merging XFDF Annotations

If annotations and form data are exported to the server using the XML command structure, then the server must implement some logic to merge the XML command into the central XFDF copy for the document. Although XFDF merging code can be easily implemented in any programming language, there are readily-made solutions available.

PDFNet SDK has the capability to merge the custom XML command from the WebViewer into an existing FDF document. To merge the XFDF on the server then, the user can host PDFNet on the server, and perform the following steps:

- fetch the central XFDF file for the document
- call FDFDoc::CreateFromXFDF() with PDFNet to create an FDF document from the XFDF document fetched
- call FDFDoc::MergeAnnots() to merge the XML command into the FDF document. This is where permission checking is done as well. For more information, please see the PDFNet documentation
- call FDFDoc::SaveAsXFDF() to save the merged FDF document as XFDF. This will be the new central XFDF copy of the document

Samples/Tutorials

Specifying the server URL for annotation loading/saving through Configuration Options

```
//In a custom js script
$.extend(ReaderControl.config,                                                    {
    serverURL : "annotationHandler.php",
    //defaultUser      is      the      Author      name      for      annotations
    defaultUser:                                                          'Guest',
});
```

Loading form data during onDocumentLoaded()

```
// Inside the BaseReaderControl constructor the current default behaviour is
// to load an external XFDF if it exists, or the internal XFDF embedded in the XOD otherwise.

    this.docViewer.setInternalAnnotationsTransform(function(originalData, callback) {
        var docIdQuery = {};
        if (me.docId !== null && me.docId.length > 0) {
            docIdQuery = {
                did: me.docId
            };
        }

        $.ajax({
            url: me.serverUrl,
```

```
            cache: false,
            data: docIdQuery,
            success: function(data) {
                if (!_.isNull(data) && !_.isUndefined(data)) {
                    callback(data);
                } else {
                    callback(originalData);
                }
            },
            error: function(jqXHR, textStatus, errorThrown) {
                /*jshint unused:false */
                console.warn("Annotations could not be loaded from the server.");
                callback(originalData);
            },
            dataType: 'xml'
        });
    });
```

Saving annotations as XFDF locally

// inside a config file call AnnotationManager.exportAnnotations();
// make use of dataURLs to download locally

```
    var am = readerControl.docViewer.getAnnotationManager();

    var xfdfString = readerControl.exportAnnotations();
    var uriContent = "data:text/xml," +
                                    encodeURIComponent(xfdfString);
    newWindow = window.open(uriContent, 'XFDF Document');
```

Exporting annotations as whole XFDF to server

// Inside AnnotationEdit.js, create an AJAX request to the server, with the data being the XFDF string
// returned by AnnotationManager.ExportAnnotations(). Note that server_url and doc_id are read in as
// the document is initially loaded

```
    if (readerControl.serverUrl == null) {
        console.warn("Not configured for server-side annotation saving.");
        return;
    }
    var am = readerControl.docViewer.getAnnotationManager();
    var xfdfString = am.exportAnnotations();
    $.ajax({
        type: 'POST',
        url: readerControl.serverUrl + '?did=' + readerControl.docId,
        data: xfdfString,
        success: function(data) {
        //Annotations were successfully uploaded to server
        },
        error: function(jqXHR, textStatus, errorThrown) {
            console.warn("Failed to send annotations to server. " + textStatus);
```

```
    },
    dataType: 'xml'
});
```

Exporting modified annotations as XML command to server

// Create an AJAX request to the server, with the data being the command
// string returned by AnnotationManager.getAnnotCommand()

```
if (readerControl.serverUrl == null) {
    console.warn("Not configured for server-side annotation saving.");
    return;
}
var am = readerControl.docViewer.getAnnotationManager();
var xfdfString = am.getAnnotCommand();
$.ajax({
    type: 'POST',
    url: readerControl.serverUrl + '?did=' + readerControl.docId,
    data: xfdfString,
    success: function(data){
    //Annotations were successfully uploaded to server
    },
    error: function(jqXHR, textStatus, errorThrown) {
        console.warn("Failed to send annotations to server. " + textStatus);
    },
    dataType: 'xml'
});
```

Sample server code to handle merging of XML command

```
<?php
include("../Lib/PDFNetPHP.php");

PDFNet::Initialize();

$con = mysql_connect("localhost","root","");
if (!$con) {
    die('Could not connect: ' . mysql_error());
} else {
    //echo "connected!";
}

if (array_key_exists('HTTP_RAW_POST_DATA', $GLOBALS)) {
  $command = $GLOBALS['HTTP_RAW_POST_DATA'];
}

if (isset($_REQUEST['did'])) {
    // check doc_id and fetch the correct xfdf
    mysql_select_db("test", $con);
    $doc_id = $_REQUEST['did'];
    $result = mysql_query("SELECT * FROM xfdf WHERE doc_id={$doc_id}");
    $row = mysql_fetch_array($result);
```

```
        $filename = $row['xfdf'];

        // call PDFNet to perform merges
        if (isset($command)) {
            $server_dir = "C:/wamp/www/server/";
            $xfdf_doc = FDFDoc::CreateFromXFDF($server_dir . $filename);
            $xfdf_doc->MergeAnnots($command);
            $xfdf_doc->SaveAsXFDF($server_dir . $filename);
        }

        // return newest copy of annots
        header("Content-type: text/xml");
        echo file_get_contents($filename);
    }
    mysql_close($con);

    ?>
```

Advanced Features

Document Encryption/Decryption

Document encryption is supported by the XOD converters and the PDFTron WebViewer is able to view these encrypted documents. Documents are encrypted with 128 bit AES (Advanced Encryption Standard), a specification from the National Institute of Standards and Technology (NIST), and is used by governments and businesses worldwide.

This allows you to implement certain forms of DRM which can be useful if you want users to only be able to view documents inside the viewer and not simply download the files to view any time outside of it. For example a web magazine viewer would likely not want users to simply download the magazines and send them to their friends. Note that the files may be able to be downloaded but they would be unviewable since they would be encrypted. Another option would be that the server doesn't store the password, the user just has to enter it in the viewer before they can view the document, effectively password protecting the document.

WebViewer with PDFNetJS supports the viewing of password protected PDF files. A dialog box will appear, prompting the user for the password.

Encryption on XOD Conversion

The process for encrypting a XOD document on conversion is simple; you just pass in a password to encrypt the document with. Depending on which XOD conversion component you are using, the way it works is:

For PDFNet with WebViewer Add-on

You can call the ToXod method with a XodConversionOption. You can specify the password in the XodConversionOptions by calling SetXodEncryptPassword.

For DocPub

You can specify the encryption password as an additional command-line argument "--

xod_encypt_password myPassword”
e.g. docpub test.pdf -f xod --xod_encrypt_password mypassword

For PWS Cloud
You can specify the password when you make your POST requests to create a document by adding it to the query parameter with the key “xodEncryptPassword”
e.g. POST https://api.pdftron.com/v1/document?xodEncryptPassword=foobar

Decryption on Client

In the client-side JavaScript you will have to make some modifications to decrypt an encrypted XOD file. When using WebViewer you need to include an ‘encryption’ property on the options object of WebViewer:

```
myWebViewer = new PDFTron.WebViewer({
    type: "html5",
    path: "",
    initialDoc: "test.xod",
    encryption: {
        p: "pass",
        type: "aes",
        error: function(msg) { alert(msg); }
    }
}, viewerElement);
```

If you are modifying ReaderControl directly then you will need to do a bit more work. Where the part retriever is created you will need to pass in two additional parameters. The first of which is the decryption function, *window.CoreControls.Encryption.Decrypt* and the second is an options object literal. The options object should have a “type” property which should be “aes”, a “password” or “p” property which should be the password used to encrypt the file and an optional “error” function. The error function is called if there is an error while attempting to decrypt the document.

```
// Sample showing how to decrypt a file.  The password could be obtained from the user
// or from a call to the server
var decrypt = window.CoreControls.Encryption.Decrypt;
var options = {
    type: "aes",
    p: "mypassword",
    error: function(msg) {
        alert(msg);
    }
};

var partRetriever = new window.CoreControls.PartRetrievers.HttpPartRetriever(doc,
true, decrypt, options);
```

Offline Mode

The WebViewer provides support for downloading XOD documents for offline viewing. Once a document has been downloaded it will be able to viewed without an Internet connection. An example use case might be a user that downloads a document, loses their connection on the train but can still continue viewing the entire document. Offline mode makes use of IndexedDB or WebSQL depending on the browser. It can also make use of the HTML5 application cache to allow for (re)loading of the page completely offline. Note that Internet Explorer 9 does not support any of these technologies and therefore is not able to support offline mode. Also note that this is not currently supported by WebViewer with PDFNetJS.

Getting Started

The HTML5 and mobile viewers have sample implementations of this which can be enabled by setting the enableOfflineMode option in WebViewer.js. The functions for offline mode are provided on the Document object. Before any other offline mode functions can be called you must call *InitOfflineDB(onComplete)*. You can pass in a callback function that will be called when the database has been initialized.

There is also an offline library sample showing how to download and view multiple documents offline. It uses the HTML5 application cache, so the page can be reloaded later on even without an Internet connection. It also uses WebViewer's *startOffline* option which allows the document to begin in offline mode and load the document from the offline database.

Downloading a Document

To begin downloading a document call its *StoreOffline(onComplete, onProgress)* function. The onComplete callback is called when the document has finished being downloaded or the download has been cancelled. The onProgress callback is called on each update in progress of the download. The fraction downloaded so far is passed in as a parameter.

To programmatically cancel an ongoing download you can call the *CancelOfflineModeDownload()* function. It will cancel any ongoing HTTP requests for parts of the document. This could be hooked up to a button as shown in *offlineReady()* in ReaderControl.js.

Enabling Offline Viewing

To actually enable offline viewing of a document you must call it's *SetOfflineModeEnabled(enabled)* function. Passing in true will enable offline mode i.e. the document will be read from the offline database. Passing in false will disable it and the document will be read from the server. You can call the *GetOfflineModeEnabled()* function to get the current state. You will probably not want to enable offline viewing until the document has finished downloading or has been downloaded previously. To check this you can call a document's *IsDownloaded()* function.

Internationalization

Internationalization is supported by WebViewer through the use of JSON language files. There is an internationalization sample provided that shows how to add a custom button that allows the user to select their preferred language.

Creating your own Translations

Several translation files are provided by default in html5/Resources/i18n. It is recommended that you create a copy of the English translation file (translation-en.json) and work from there. Your custom file should have a file name in the form translation-*languagecode*.json and simply needs to be placed in the same directory.

To edit the translation file you need to change the values to the right of the colon. For example to edit the “Full Screen” translation:

```
"controlbar": {
    "fullScreen": "My New Translation",
    …
}
```

A couple of the translation values have text that starts and ends with two underscores:

```
"pageNumber": "Page __number__",
```

In this case __number__ is a variable and will have its value substituted in at run time (eg Page 5). Only the non-variables should be changed for these translations (eg. "Página __number__").

4. Native Application Integration

While the HTML5 WebViewer was designed for web browsers, it can also be embedded directly in native mobile applications. Viewing documents directly from the file system without an Internet connection can be achieved this way. Note that WebViewer with PDFNetJS does not support this.

For Android, iOS and WinRT, WebViewer can be embedded in a WebView or UIWebView control. In all cases, WebViewer will be able access documents directly on the device's file system.

Android
A sample of Android integration is included in the WebViewer package. It features a custom ContentProvider (LocalFileContentProvider) that reads local XOD documents and delivers the required parts to the WebViewer. Essentially, it simulates a web server's byte range request capabilities to provide content to the viewer only when it is needed.
On the WebViewer side, there is a part retriever (AndroidContentPartRetriever) that will handle making the requests to the ContentProvider.

iOS & WinRT
The iOS and WinRT integration samples use special part retrievers which communicate with the app's UIWebView/WebView. Similar to Android they simulate a byte range request and the requested range is passed to the app. The apps read the bytes directly from the file and base64 encode the data before passing it back to JavaScript through a callback.

5. Appendix

Resources

For more resources, please see our PDFTron WebViewer support forum. WebViewer developers frequently monitor the forum and answer questions.
https://groups.google.com/forum/#!forum/pdfnet-webviewer

Need more support?
Report a problem here: http://www.pdftron.com/support/reportproblem.html

Or email support@pdftron.com directly to get in touch with us

HTML5 ReaderControl Hash Parameters

When the HTML5 viewer, ReaderControl, is used by itself outside of WebViewer.js, hash parameters can be used to control the viewer. Boolean parameters can be specified as true/false, yes/no or 0/1. Below is the list of parameters that are available:

a – A boolean parameter that specifies whether annotations are enabled or not. If 'a' is false, annotations will not be displayed on the document, and no annotations-related operations can be performed. See also: Toggle Annotations

admin – A boolean parameter that specifies whether the current user is an administrator. An admin user can perform add/modify/delete operations on annotations without restrictions. See also: Permission Checking

config – The URL to a JavaScript file that contains configuration options and customizations to the ReaderControl. The config file can be from a different host than the ReaderControl.

d – The string that contains the path to the document to be displayed on WebViewer. This is the relative path to the location which hosts ReaderControl.html.

did – The ID of the document to be displayed on WebViewer. It is a string without any special restrictions. This ID, assigned by a server, can be used to fetch the correct XFDF annotations file if there are multiple documents stored on server. On the other hand, 'did' can also be used as a session token, in order to authenticate the client user.

filepicker - A boolean parameter that specifies whether to show a local file picker button in the toolbar. This is only applicable when viewing PDF documents.

offline – A boolean parameter that specifies whether offline mode buttons are shown or not. There is a button to download the document and a button to toggle offline mode on or off.

readonly – A boolean parameter that specifies whether readonly mode is enabled or not. Annotations cannot be modified in any way in readonly mode. See also: Readonly Mode

server_url – The URL to the server script that handles AJAX requests sent from the WebViewer client. For example, these can be requests to fetch the XFDF annotations file for the displayed document.

startOffline – A boolean parameter that specifies whether the viewer should start in offline mode or not. The viewer must start in offline mode if the page will be viewed without a network connection.

streaming – A boolean parameter that specifies whether or not to use document streaming. Please note that streaming refers to serving the XOD document AS it is converting. Using streaming mode degrades performance and should only be used if XOD conversions are done on-the-fly or if the XOD file host does not support byte range requests.

user – A string that specifies the current user of the WebViewer. This is used to record the author of any new annotations added to the document, as well as perform user permission checks with regards to operations on existing annotations on the document.