



# WebView Developer Guide

*version 1.5.0*

## Table of Contents

- [Introduction](#)
  - [Audience and Scope](#)
  - [WebView Components](#)
    - [Client-side Viewer Application](#)
    - [Platforms Supported](#)
  - [Document Conversion Component](#)
    - [PDFNet Cloud API](#)
    - [PDFNet SDK with WebViewer Add-on](#)
    - [DocPub Command-line Converter](#)
  - [Your Server Component](#)
    - [WebViewer and XOD Hosting](#)
    - [Annotation Handling](#)
- [WebViewer: The Basics](#)
  - [Using JavaScript with WebViewer](#)
  - [XOD Conversion](#)
- [Customization Framework](#)
  - [Configuration Options](#)
  - [External Configuration File](#)
  - [Pass Custom Data through WebViewer](#)
  - [Customization with PDFNet Cloud API](#)
- [Annotations and Forms](#)
  - [Markup Annotations](#)
    - [Permission Checking](#)
    - [Read-only Mode](#)
    - [Toggle Annotations](#)
- [Forms](#)
  - [Supported Field Widget Types](#)
    - [Button Fields](#)
    - [Text Fields](#)
    - [Choice Fields](#)
  - [Supported Form Actions](#)
    - [Submit Form Actions](#)
    - [Reset Form Actions](#)
    - [JavaScript Actions](#)
    - [Hide Actions](#)

[Processing Annotations and Forms](#)

[Loading Annotations and Form Fields](#)

[Exporting Annotations and Form Fields](#)

[Merging XFDF Annotations](#)

[PDFNet SDK](#)

[Samples/Tutorials](#)

[Advanced Features](#)

[Document Encryption/Decryption](#)

[Encryption on XOD Conversion](#)

[Decryption on Client](#)

[Offline Mode](#)

[Getting Started](#)

[Downloading a Document](#)

[Enabling Offline Viewing](#)

[Native Application Integration](#)

[Internationalization](#)

[Creating your own Translations](#)

[Appendix](#)

[HTML5 ReaderControl Query Parameters](#)

## 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 HTML5 and Mobile viewers. Customizations on the Silverlight and Flash viewers are not discussed here.

### 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
3. Your own server for document hosting and annotation handling (optional)

### Client-side Viewer Application

The PDFTron WebViewer is a solution that employs several different client-side application technologies to deliver consistent and reliable document viewing across all major platforms and environments. Under the hood, the PDFTron WebViewer consists of several client-side applications implemented with different browser technologies (HTML5, mobile, Silverlight and Flash). 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.

Table 1. Feature Support by WebViewer Technology

	HTML5	Mobile	Silverlight	Flash
Document Viewing	✓	✓	✓	✓
Annotations	✓	✓	✓	
Forms	✓	✓		
Customization Framework	✓	✓		
Offline Mode	✓	✓		
Encrypted XOD files	✓	✓		✓



### *Platforms Supported*

- HTML5 (any modern browser with Canvas support; e.g. IE9+, Chrome, FireFox, Safari, Opera).
- Silverlight 5 (Windows, Mac, Windows Phone 7 – Mango)
- Flash (Windows, Linux, Mac, Mobile).
- 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.

### **Document Conversion Component**

The WebViewer solution supports viewing a wide range of document file formats by converting all documents to the XOD format.

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

#### *PDFNet Cloud API*

This 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 Cloud API server, along with a hosted version of WebViewer. This is the easiest and most cost-efficient way of converting documents.

For more information, see <http://www.pdftron.com/pdfnet/cloud>.

#### *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 features here: 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.

For more information, see <http://www.pdftron.com/pdfnet>

## *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.

Table 2. XOD Conversion Features by Converter Types

	PDFNet SDK	DocPub CLI	PDFNet Cloud API
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 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 XOD Hosting

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

## **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.

## WebView: The Basics

For developers wishing to take advantage of the universal document viewing technology to support as many clients as possible, using the WebView.js wrapper is the way to go. WebView.js makes it easy to embed a viewer control in an HTML page with JavaScript. WebView.js detects browser and platform support to load the best viewer technology.

Developers who do not need cross-platform support may choose to use the WebView viewer technologies independently. For example, developers who only need to support modern browsers and mobile devices can use the HTML5 viewer directly. In this case, please see the API reference that is included with the viewer directly.

## Using JavaScript with WebView

To create your own WebView 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 type="text/javascript" src="jquery-1.7.2.min.js"></script>
<script type="text/javascript" src="WebView.min.js"></script>
```

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

```
<script type="text/javascript" src="silverlight/Silverlight.js"></script>
<script type="text/javascript" src="flash/swfobject.js"></script>
<script type="text/javascript" src="flash/FABridge.js"></script>
<script type="text/javascript" src="flash/FlashUtils.js"></script>
<script type="text/javascript" src="WebView.js"></script>
```

WebView.js depends on jQuery, and helper scripts for Silverlight and Flash. The Silverlight and Flash scripts are optional if you do not plan on using those technologies. This will allow you to use the WebView class in your JavaScript code.

3. Create a `<div>` 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 WebView.



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

The script above will create and render a PDFNet WebViewer control on the HTML page as a child of the provided <div> element. The code specifies that the HTML5 viewer be attempted first. If the script detects that canvas is not supported by the client's web browser then the Silverlight plug-in will be loaded next. If Silverlight is not present the script will finally try using Flash. Regardless of the viewer technology, the viewer created will load the document as specified by initialDoc immediately.

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

5. It is now possible to use the "myWebViewer" variable to invoke ReaderControl's methods. For example, the following code demonstrates how to load a document:

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

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

```
$(viewerElement).bind('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 that is shared across all viewer technologies\*.

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/pdfnet/webviewer/jsdoc/>

\*Please note that the HTML5 Mobile viewer is optimized for mobile devices and therefore cannot be controlled through WebViewer.js. If the 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).



## XOD Conversion

Once you have successfully created your own WebViewer instance, you will want to view your own documents. To do so, you will need to convert your documents to the XOD format. Please choose one of the three XOD converters available: <http://www.pdftron.com/pdfnet/webviewer/download.html>

### *PDFNet Cloud API*

With Cloud API, 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/pdfnet/cloud/snippets.html>

### *PDFNet SDK with WebViewer Add-on*

With the full-featured PDF SDK, users can convert their PDF 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>. For sample code to convert PDF documents to XOD documents, see <http://www.pdftron.com/pdfnet/samplecode.html>.

### *DocPub command-line converter*

DocPub is a command-line tool that allows conversion from PDF to XOD.

The command is docpub [<options>] file. Here is a list of common options used:

--file... arg	A list of folders and/or file names to process.
-s [ --subfolders ]	Process all subfolders.
-f [ --format ] arg	The output format. The following is a list of supported export formats:



pdf  
xps (Print XPS)  
xod (Web optimized XPS)

- o [ --outdir ] arg      The output directory. The folder can be relative to the current working folder. If the folder does not exist DocPub will attempt to create the required path. If this parameter is not specified all files will be saved relative to the current working folder.
- fname arg      The explicit name for the converted file. The parameter is used only when converting one file at a time. The output file will be stored in the 'outdir' folder.
- extension arg      The file extension which filters the files to be processed by the specified type.
- p [ --pass ] arg      The password to process the input file. The password is required only when the input document is encrypted.
- d [ --dpi ] arg (=92)      For conversions to .xod only. The output resolution, from 1 to 1000, in Dots Per Inch (DPI) at which to render elements which cannot be directly converted. The higher the DPI, the larger the image. The default resolution is 92 DPI.

For example, here is the command used to batch convert a directory of PDF files to Web optimized XPS (XOD):

```
docpub -f xod --subfolders --extension pdf mypdfdir
```

For more information, please see the usage documentation included in the DocPub CLI package.

## Customization Framework

The HTML5 WebViewer was designed for easy customization of the user interface and custom features. The ReaderControl control offers a configuration file. While it is possible to customize the Silverlight and Flash viewers as well, it is not compatible with the customization framework and will not be discussed in this document.

### 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 string.

To find out more about the options available, please see the “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. When the query parameter “config” is provided, the viewer will download this script dynamically.

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 changing the theme 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 PDFNet Cloud API

If you choose to use PDFNet Cloud API for your XOD conversions, your documents will 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 Cloud API 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. Please note that the annotation feature is available for HTML5, mobile, and silverlight viewers, while form support is only available for the HTML5 and mobile viewers.

### 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 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 up 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. If using ReaderControl directly then it can be enabled by passing 'true' into the 'readonly' query parameter. 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 popup messages 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 Annotation Panel, the user can toggle between showing and hiding all the annotations on the currently displayed document. This is similar to changing the query parameter 'a' which decides whether annotations are enabled, but the ability is dynamic, and the user can switch between having annotations on or off quickly while reading the document.

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 or the annotations list 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 PDFTron WebViewer to re-create the field widgets 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 common form actions, such as submit form action, JavaScript action

## 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 contain 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 of the PDF document. When the widget is clicked, `eval()` is called to execute the stored JavaScript.

### *Hide Actions*

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

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

The WebViewer works by loading XOD files, therefore the user needs to convert the PDF source document into a XOD document first. During the convert 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 the PDFTron 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'. If using ReaderControl directly then the query parameter 'a' must be set to 1.

While the user can provide an external XFDF file to load annotations from in 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

The 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.

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">
  <doc_id>doc_id</doc_id>
  <fields />
  <add />
  <modify />
  <delete />
</xfdf>
```

The doc\_id element contains the document ID of the opened document. 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

The 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 whose path is provided in the “config” query string
$.extend(ReaderControl.config,
    {
        serverURL : "annotationHandler.php",
        //defaultUser is the Author name that is used to create
        defaultUser: 'Guest',
    });
```

### *Loading form data during onDocumentLoaded()*

```
// Inside ReaderControl.js and the function onDocumentLoaded(), the current default behaviour is
// to load an external XFDF if it exists, or the internal XFDF embedded in the XOD otherwise.
// the 'externalAnnotsExist' parameter is used to decide whether the internal XFDF is loaded.
// Note that server_url and doc_id are read in as the document is initially loaded.
```

```
var queryData = {};
if(this.doc_id != null && this.doc_id.length > 0) {
```

```
        queryData = {
            'did': this.doc_id
        };
    }

    $.ajax({
        url: this.server_url + '?did=' + this.doc_id,
        cache: false,
        data : queryData,
        success: function(data) {
            if(data != null) {
                am.externalAnnotsExist = true;
                am.LoadAnnotations(data);
            }
        },
        error: function(jqXHR, textStatus, errorThrown) {
            console.warn("Annotations could not be loaded from the
                           server.");
            am.externalAnnotsExist = false;
        },
        dataType: 'xml'
    });
```

#### *Saving annotations as XFDF locally*

// Inside AnnotationPanel.js, create an AnnotationManager and call  
// AnnotationManager.SaveAnnotations(); make use of dataURLs to download locally

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

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

#### *Exporting annotations as whole XFDF to server*

// Inside AnnotationPanel.js, create an AJAX request to the server, with the data being the XFDF string  
// returned by AnnotationManager.SaveAnnotations(). Note that server\_url and doc\_id are read in as  
// the document is initially loaded

```
if(readerControl.server_url == null) {
    console.warn("Not configured for server-side annotation saving.");
    return;
}
var am = readerControl.docViewer.GetAnnotationManager();
var xfdfString = am.SaveAnnotations();
$.ajax({
    type: 'POST',
    url: readerControl.server_url + '?doc_id=' + readerControl.doc_id,
    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*

// Inside AnnotationPanel.js, create an AJAX request to the server, with the data being the command string returned by AnnotationManager.GetAnnotationCommand(). Note that server\_url and doc\_id are read in as the document is initially loaded

```
if(readerControl.server_url == null) {
    console.warn("Not configured for server-side annotation saving.");
    return;
}
var am = readerControl.docViewer.GetAnnotationManager();
var xfdfString = am.GetAnnotationCommand();
$.ajax({
    type: 'POST',
    url: readerControl.server_url+'?doc_id='+readerControl.doc_id,
    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.

### Encryption on XOD Conversion

The process for encrypting a XOD document on conversion is simple, you 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 PDFNet DocPub*

You can specify the encryption password as an additional command-line argument "--xod\_encrypt\_password myPassword"

e.g. docpub test.pdf -f xod --xod\_encrypt\_password mypassword

*For PDFNet Cloud API*

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,flash",
  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" 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",
  password: "pass",
  error: function(msg) {
    alert(msg);
  }
};
```

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

## Offline Mode

The WebViewer provides support for downloading 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. Note that Internet Explorer 9 does not support either of these technologies and therefore is not able to support offline mode.

## Getting Started

The HTML5 and mobile viewers have sample implementations of this which can be enabled by setting the `enableOfflineMode` option in `WebViewer.js` or by setting the query parameter “offline” to 1 directly in `ReaderControl`. 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.

## 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 its *`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.

## 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.

For Android and iOS, the WebViewer can be embedded in a WebView and UIWebView control, respectively. In both cases, the 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 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.

## Internationalization

Internationalization is supported by the HTML5 WebViewer through the use of JSON language files. There is an internationalization sample provided in the HTML5 samples section 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 simply 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__`").

## Appendix

### ***HTML5 ReaderControl Query Parameters***

When the HTML5 viewer, ReaderControl, is used by itself outside of WebViewer.js, query string parameters can be used to control the viewer. 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

*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.

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

*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.

*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

*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

*offline* – A boolean parameter that specifies whether offline mode is enabled on or not. In offline mode the user can download the document for offline viewing.

*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.

*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.