

PROJECT IT-SECURITY

PDF-OVER AND HTML5

REPORT & DOCUMENTATION

THOMAS FELBER 1031194

Contents

1 Report

1.1	Introduction	
1.2	Status Quo	
1.3	Motivation and Goal	
1.4	Solution	

2 Documentation

2.1	Project Files	
2.2	Signature placement	
2.3	Entry Points to the PDF.js standalone application	
2.4	Overriding/Extending behaviour of the sign button in PDF.js standalone	
2.5	Limitations	

1 Report

1.1 Introduction

PDF-AS is a software library, developed in JAVA, that makes use of advanced and qualified certificates, to allow users to electronically sign PDF documents. One of the core components of *PDF-AS* is its web module *PDF-AS Web*. *PDF-AS Web* is built upon *PDF-AS* and establishes a web interface that can be used to sign PDF documents but also can be used by external applications to embed PDF signatures.

1.2 Status Quo

Currently, the *PDF-AS Web* online presence is quite basic. It offers different

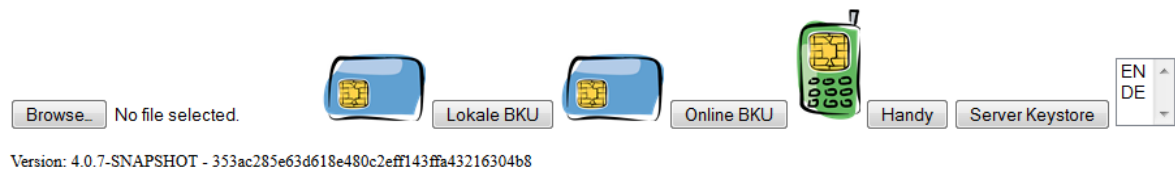


Figure 1: Current state of the *PDF-AS Web* online presence (https://demo.egiz.gv.at/demportal-pdf_as/)

methods to sign PDF documents, however isn't really visually appealing yet and doesn't offer any extra features like PDF preview or manual signature placement.

1.3 Motivation and Goal

Since *PDF-AS Web*'s online presence is still that slim, it is of concern to improve its current state. Therefore, in the course of this project, several enhancements to the website are being made. The whole website gets a visual overhaul using HTML5 in combination with CSS3. In terms of additional features, Mozilla Labs' *PDF.js* library is used to make previewing PDF documents possible. Also new features that come along with HTML5, such as drag and drop file selection, are being taken advantage of. That is, the goal of this project can be summarized like that:

- Use HTML5 + CSS3 and perform a visual overhaul on the website
- Introduce new features like a PDF rendering and drag and drop file selection
- Let the user manually place signatures

1.4 Solution

Visual overhaul

As already mentioned in Section 1.3, HTML5 + CSS3 was used to recreate *PDF-AS Web's* web interface. The new website consists of several different parts and is illustrated in Figure 2. The *File Selector* section is here to let a user select the PDF document that he wants to sign. He can either choose to select the PDF file via the traditional HTML file selector element (i.e. press on "Browse...") or just drag and drop the PDF file from his file manager into the dropzone (upper part of the *File Selector* section). Via the *Sign Method* section, a user can choose

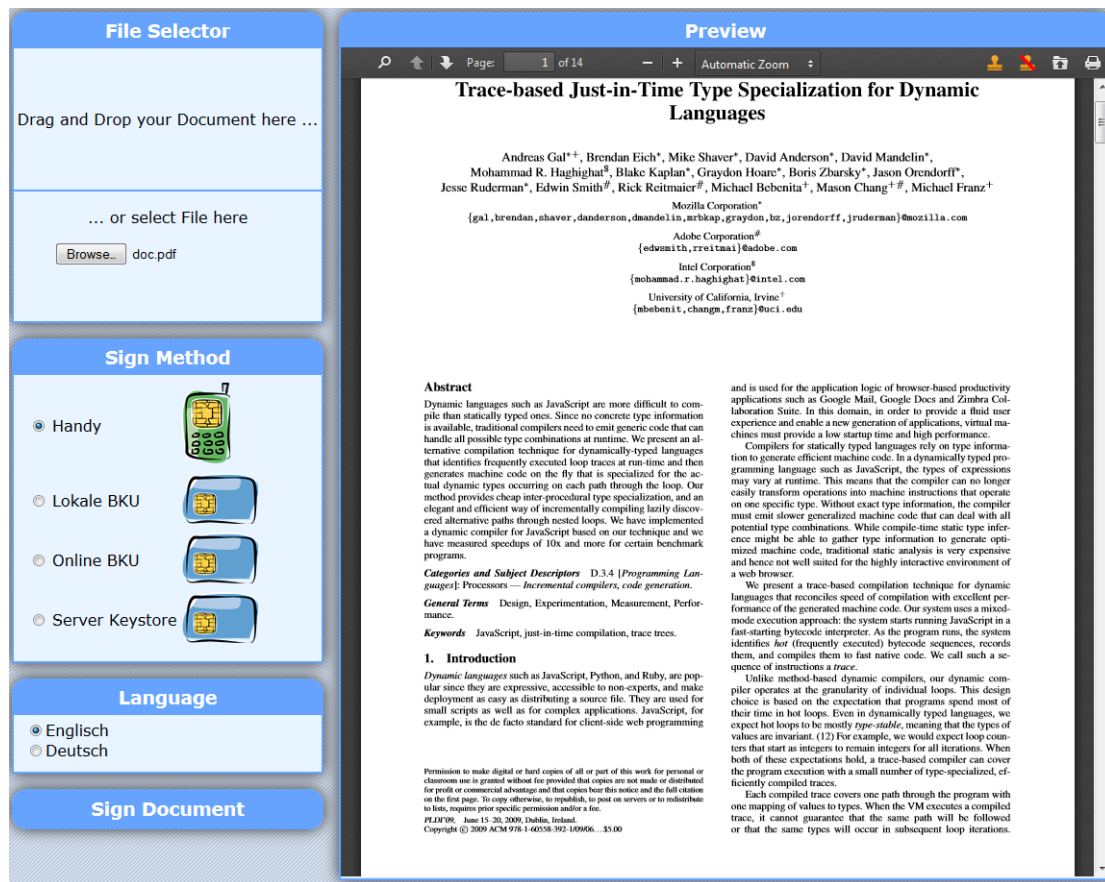





Figure 2: Updated *PDF-AS Web* online presence

between different methods to actually sign a previously selected PDF document. Available options are *Handy*, *Lokale BKU*, *Online BKU* and (if enabled on the web server) *Server Keystore*. Depending on the choice, the user gets redirected

to a different service that requires some kind of authentication and then signs the PDF document with the user's electronic signature. Within the *Language* section, a user has the option to select which language he wants the signature to be in. Currently only German and English are supported. Below the *Language* section, there is a button that starts the sign process if it is clicked.

A main component of the website is the *Preview* section. Within that, a user gets a visual representation of the PDF document which he has previously selected. He can scroll through the different pages of the document, adjust the zoom, open a different file, print the file, place a signature etc. .

PDF renderer (PDF.js) and signature placement

In order to render the user selected PDF document, Mozilla Labs' *PDF.js* library is being used. Within the *Preview* section in Figure 2, a customized instance of the *PDF.js* renderer can be seen. Some of the customization that have been made to *PDF.js* affect the user interface. Various buttons and features that are not really necessary have been removed (like downloading the currently displayed PDF document, switching to presentation mode ...) and other important features have been added. Part of those features are the two little buttons . These buttons are not available in the stock *PDF.js* implementation. They allow the user to manually perform the signature placement. On pressing the  button, a preview of the signature block is being shown on the page the user is currently on, representing the position of the signature block after the document has been signed. See Figure 3 for an illustration. The user can move the signature block around with his computer mouse and drag it wherever he wants it to be. If he changes his mind and does not want to place it manually anymore, he just has to interact with the  button which removes the placed signature block. This action implies that the signature block is being placed automatically by the application.

PDF renderer (PDF.js) as standalone application

Another way to interact with *PDF-AS Web* is to just use the PDF renderer alone. This might be useful in scenarios where external applications make use of *PDF-AS Web*, e.g. a PDF document is uploaded via SOAP upload to the web server by an external application, then the PDF renderer can be used to show a quick preview of the document. For that particular case, where the *PDF.js* instance is used as standalone application, the user interface is further extended, see Figure 4.

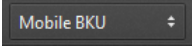

Via the drop-down menu  the user can switch between the different sign methods directly within the *PDF.js* instance and with the  button, the



Figure 3: *PDF-AS* Web PDF renderer with manually placed signature block

sign process can be started. Furthermore, the standalone version of the *PDF.js* is customized to support different HTML get request parameters, such as *file*, *pdfurl* and *connector*. Example usage of those parameters shall be shown in Section 2.3

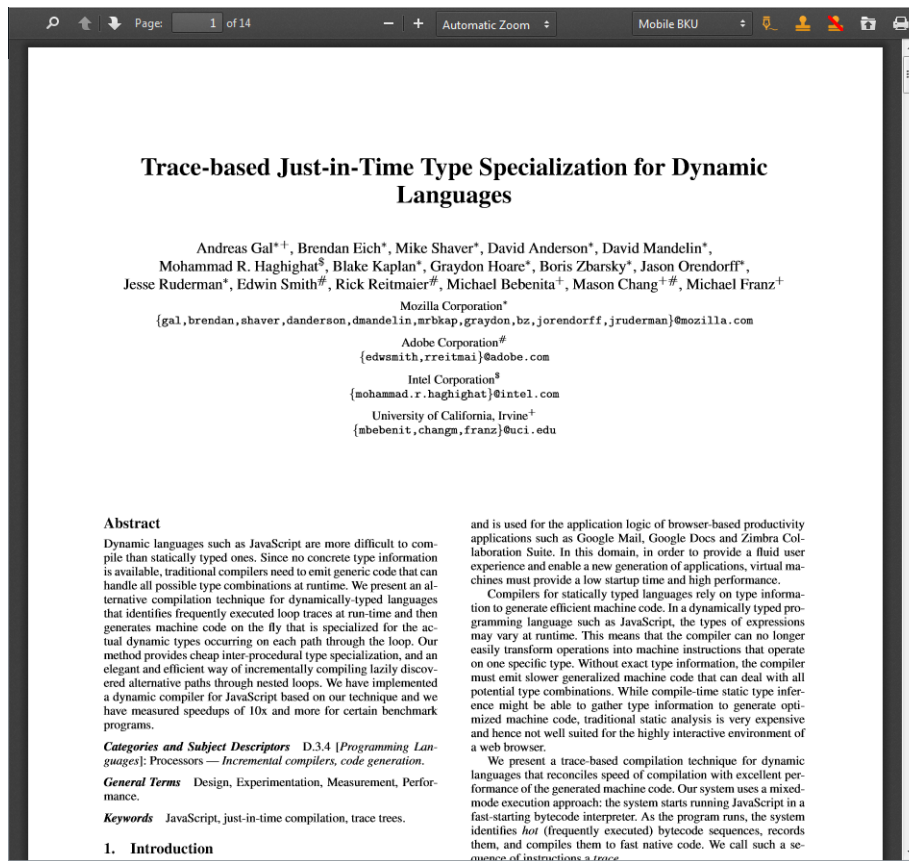


Figure 4: *PDF-AS Web*, PDF renderer only

2 Documentation

2.1 Project Files

The main files and folders worked with during this projects are:

- ./pdf-as-web/
 - index.jsp /* Start page of *PDF-AS Web*'s online presence */
- ./pdf-as-web/assets/css/
 - style.css /* Style sheet for *PDF-AS Web*'s online presence */
- ./pdf-as-web/assets/js/
 - dragNdrop.js /* JavaScript for *PDF-AS Web*'s online presence */

- ./pdf-as-web/assets/js/pdf.js/web/
 - viewer.html /* Customized HTML template for the *PDF.js* viewer */
 - viewer.js /* Customized version of *PDF.js*' viewer.js */
 - app.js /* JavaScript that works within PDJ.js' viewer.html */

dragNdrop.js

This JavaScript file is used within the **index.jsp** file and is responsible for handling things like

- drag and drop file selection
- event handling
- opening the PDF preview for a selected PDF document
- performing requests to the sign servlet if a users presses the sign button

app.js

This JavaScript file is used within the **viewer.html** file and is responsible for handling things like

- using *PDF.js* functionality to open and display PDF documents
- manual signature placement
- event handling
- managing the behaviour if *PDF.js* is used as standalone application

viewer.js

This JavaScript file is a main component of the *PDF.js* library and has been slightly customized to fit into this project. The changes made to this file are certainly good to know if the application wants to get updated to a newer *PDF.js* versions, hence, they shall be denoted as follows:

The function *webViewerLoad(evt)* starting at line 6668 has been altered, to display the user selected PDF document as soon as *PDF.js* is initialized.


```
1 function webViewerLoad(evt) {
2   //PDFViewerApplication.initialize().then(webViewerInitialized);
3   PDFViewerApplication.initialize().then(function() {
4     webViewerInitialized();
5     displayPdf()
6   });
7 }
```

State: webViewerLoad after the change

```
1 function webViewerLoad(evt) {
2   PDFViewerApplication.initialize().then(webViewerInitialized);
3 }
```

State: webViewerLoad before the change

The function *webViewerChange(evt)* starting at line 7006 has been altered, such that, if a user opens a new file via the *PDF.js* user interface, the file variable within **app.js** also gets updated (needs to be done to make sure that the sign function in **dragNdrops.js** and **app.js** works).

```
1 window.addEventListener('change', function webViewerChange(evt) {
2   ...
3   var file = files[0];
4
5   //ADDED
6   setFile(file);
7
8   if (!PDFJS.disableCreateObjectURL &&
9     ...
10 }
```

State: webViewerChange after the change

```
1 window.addEventListener('change', function webViewerChange(evt) {
2   ...
3   var file = files[0];
4
5   if (!PDFJS.disableCreateObjectURL &&
6     ...
7 }
```

State: webViewerChange before the change

The function *webViewerInitialized()* starting at line 6676 has been altered in a way that *PDF.js* does not handle the HTML get request parameter *file* anymore. The handling of that parameter is done in **app.js**.

```

1 function webViewInitialized() {
2     ...
3     /*if (file && file.lastIndexOf('file:', 0) === 0) {
4         // file:-scheme. Load the contents in the main thread because QtWebKit
5         // cannot load file:-URLs in a Web Worker. file:-URLs are usually loaded
6         // very quickly, so there is no need to set up progress event listeners.
7         PDFViewerApplication.setTitleUsingUrl(file);
8         var xhr = new XMLHttpRequest();
9         xhr.onload = function() {
10            PDFViewerApplication.open(new Uint8Array(xhr.response), 0);
11        };
12        try {
13            xhr.open('GET', file);
14            xhr.responseType = 'arraybuffer';
15            xhr.send();
16        } catch (e) {
17            PDFViewerApplication.error(mozL10n.get('loading_error', null,
18                'An error occurred while loading the PDF.'), e);
19        }
20        return;
21    }
22
23    if (file) {
24        PDFViewerApplication.open(file, 0);
25    }*/
26 }

```

State: webViewInitialized after the change

```

1 function webViewInitialized() {
2     ...
3     if (file && file.lastIndexOf('file:', 0) === 0) {
4         // file:-scheme. Load the contents in the main thread because QtWebKit
5         // cannot load file:-URLs in a Web Worker. file:-URLs are usually loaded
6         // very quickly, so there is no need to set up progress event listeners.
7         PDFViewerApplication.setTitleUsingUrl(file);
8         var xhr = new XMLHttpRequest();
9         xhr.onload = function() {
10            PDFViewerApplication.open(new Uint8Array(xhr.response), 0);
11        };
12        try {
13            xhr.open('GET', file);
14            xhr.responseType = 'arraybuffer';
15            xhr.send();
16        } catch (e) {
17            PDFViewerApplication.error(mozL10n.get('loading_error', null,
18                'An error occurred while loading the PDF.'), e);
19        }
20        return;

```

```
21 }
22
23 if (file) {
24     PDFViewerApplication.open(file, 0);
25 }
26 }
```

State: `webViewerInitialized` before the change

2.2 Signature placement

The signature placement functionality is implemented within the `app.js` file. The main functions responsible for that are

```
1 function placeSignature(evt) {
2     ...
3 }
```

Purpose: Place a signature preview at the top left of the current page of the PDF document. The signature preview is created by the `visblock` servlet. Information about what page the signature should be placed on is extracted from `PDF.js`

```
1 function makeSignatureDraggable(signature) {
2     ...
3 }
```

Purpose: Uses `jquery.ui` to make a previously added signature preview draggable within the page it appears in.

```
1 function updateSignaturePosition(signature) {
2     ...
3 }
```

Purpose: Keeps track of the position of the signature within the page it appears in. This is needed because the `sign` servlet expects the position of the signature block as request parameter if the signature is placed manually.

```
1 function removeSignature() {
2     ...
3 }
```

Purpose: Remove a previously placed signature block.

2.3 Entry Points to the PDF.js standalone application

As already mentioned in Section 1.4, there exist HTML get request parameters that can be appended to the URL to control the behaviour of the *PDF.js* standalone application. The set of parameters consists of *file*, *pdfurl* and *connector*. The *file* parameter can be used to display a PDF document that is already available on the web server. For example if someone makes a request to <http://localhost:8080/pdf-as-web/assets/js/pdf.js/web/viewer.html?file=/pdf-as-web/assets/test.pdf> then the file located at */pdf-as-web/assets/test.pdf* is fetched from the server and rendered with *PDF.js*.

It is also possible to combine the *file* parameter with the *connector* parameter. With the connector parameter, it is possible to already preselect a particular sign method. So, if someone requests for example <http://localhost:8080/pdf-as-web/assets/js/pdf.js/web/viewer.html?file=/pdf-as-web/assets/test.pdf&connector=bku> then the file located at */pdf-as-web/assets/test.pdf* is fetched from the server and rendered with *PDF.js* and the sign method will be preselected to *bku*, which means that Local BKU should be used as sign method. A list of the currently supported sign methods is available in Table 1.

Table 1: Values of connector and what it preselects

Connector parameter	Preselects
bku	Local BKU
onlinebku	Online BKU
mobilebku	Handy

Another way to use the *PDF.js* standalone application is to call it via the *pdfurl* parameter and the *connector* parameter. This will immediately start the sign process on the PDF document given by the *pdfurl* and the chosen sign method given by *connector*. E.g. a request to <http://localhost:8080/pdf-as-web/assets/js/pdf.js/web/viewer.html?pdfurl=http://kmmc.in/wp-content/uploads/2014/01/lesson2.pdf&connector=bku> starts the sign process for the PDF document available by the url *http://kmmc.in/wp-content/uploads/2014/01/lesson2.pdf* using the *Online BKU* method.

Of course, *PDF.js* standalone can also be used without appending any HTML get request parameters. In that case, the user gets served with the plain *PDF.js* viewer without any PDF documents being open. To open a PDF document, the user can use the "open File" button of the menu bar. All the other features like signature placement and choosing a sign method are naturally available as well.

2.4 Overriding/Extending behaviour of the sign button in PDF.js standalone

If the sign button is clicked, the function sign

```
1 function sign(statusObj) {
2   var file = statusObj.getFile();
3   if(!file) {
4     alert("No File Opened");
5     return;
6   }
7   var fd = new FormData();
8   fd.append("source", "internal");
9   fd.append("connector", global_status.getConnector());
10  fd.append("pdf-file", global_status.getFile());
11
12  if(isSignaturePlaced()) {
13    fd.append("sig-pos-x", global_status.getSignature().posx);
14    fd.append("sig-pos-y", global_status.getSignature().posy);
15    fd.append("sig-pos-p", global_status.getSignature().page);
16  }
17
18  $.ajax({
19    url: "/pdf-as-web/Sign",
20    data: fd,
21    processData: false,
22    contentType: false,
23    type: "POST",
24    success: function(response) {
25      $("#fade").remove();
26      $("#popup").remove();
27      var fade_div = "<div id='fade' class='black_overlay'></div>";
28      var popup_div = "<div id='popup' class='white_content'><a href='javascript
29      :void(0)' id='closelink'>Close</a><div id='resp'></div></div>"
30      $("#body").append(fade_div);
31      $("#body").append(popup_div);
32      $("#resp").html(response);
33      $("#closelink").bind("click", function(evt) {
34        $("#fade").remove();
35        $("#popup").remove();
36      });
37    });
38 }
```

within the app.js file is being executed. This function expects a status object as its only argument.

So far, the status object is a global object within the `app.js` file and encapsulates information about which connector is selected and which file is selected, the filepath that was provided by the `file` HTML get request parameter, the `pdfurl` that was given by the `pdfurl` HTML get request parameter, and a not yet used `redirect url`.

The sign function can for instance be modified in a way such that the function acts differently for status objects with different values.

2.5 Limitations

Due to some technologies that are being used in this project, not all browsers are fully supported. Especially older versions of the Internet Explorer are not supported at all. Table 2 gives a brief overview about which browsers are supported by the `PDF.js` library.

Table 2: `PDF.js` supported browsers (src: <https://github.com/mozilla/pdf.js/wiki/Frequently-Asked-Questions>)

Browser	Supported	Notes
Firefox Stable	yes	-
Chrome Stable	yes	-
Opera Stable	yes	-
Android	limited	Android's Web Browser version 4.0 or below lacks a number of features or has defects, e.g. in typed arrays or HTTP range requests
Safari	limited	Safari (desktop and mobile) lacks a number of features or has defects, e.g. in typed arrays or HTTP range requests
IE10+	limited	IE 10 or above may lack of features or may have defects.
IE9	limited	IE9 lacks a number of features and most notably typed arrays which causes subpar performance
<=IE8	NO	IE8 and below are missing too many features to be supported.