Project IT-Security

PDF-Over and HTML5

Report & Documentation

Thomas Felber 1031194

Contents

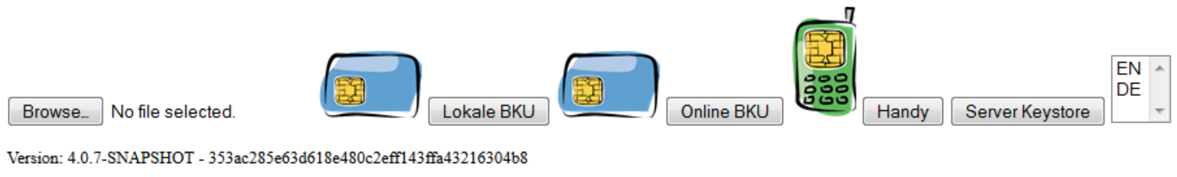
1 Report

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 it's 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



Version: 4.0.7-SNAPSHOT - 353ac285e63d618e480c2eff143ffa43216304b8

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

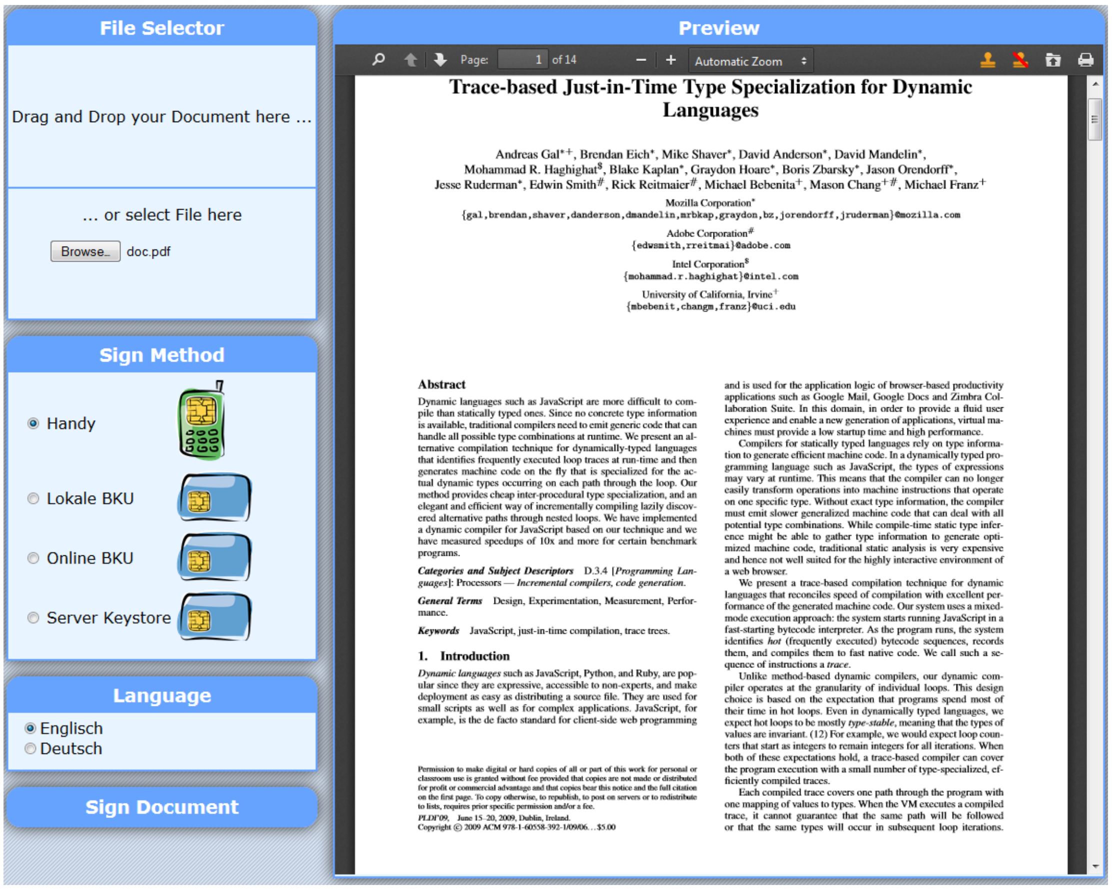
Since *PDF-AS Web's* online presence is still that slim, it is of concern to improve it's current state. Therefore, in the course of this project, several enhancements to the website are being made. The whole website gets a visual overhaul using HTML5 in combination with CSS3. In terms of additional 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) within the *File Selector* section. Via the *Sign Method*



Trace-based Just-in-Time Type Specialization for Dynamic Languages

Andreas Gal*+, Brendan Eich*, Mike Shaver*, David Anderson*, David Mandelin*, Mohammad R. Haghighat$, Blake Kaplan*, Graydon Hoare*, Boris Zbarsky*, Jason Orendorff*, Jesse Ruderman*, Edwin Smith#, Rick Reitmaier#, Michael Bebenita+, Mason Chang+#, Michael Franz+

Mozilla Corporation*
{gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Adobe Corporation#
{edwsmith,rreitmai}@adobe.com

Intel Corporation$
{mohammad.r.haghighat}@intel.com

University of California, Irvine+
{mbebenit,changm,franz}@uci.edu

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of 10x and more for certain benchmark programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — *Incremental compilers, code generation.*

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, just-in-time compilation, trace trees.

1. Introduction

Dynamic languages such as JavaScript, Python, and Ruby, are popular since they are expressive, accessible to non-experts, and make deployment as easy as distributing a source file. They are used for small scripts as well as for complex applications. JavaScript, for example, is the de facto standard for client-side web programming and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code, traditional static analysis is very expensive and hence not well suited for the highly interactive environment of a web browser.

We present a trace-based compilation technique for dynamic languages that reconciles speed of compilation with excellent performance of the generated machine code. Our system uses a mixed-mode execution approach: the system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies *hot* (frequently executed) bytecode sequences, records them, and compiles them to fast native code. We call such a sequence of instructions a *trace*.

Unlike method-based dynamic compilers, our dynamic compiler operates at the granularity of individual loops. This design choice is based on the expectation that programs spend most of their time in hot loops. Even in dynamically typed languages, we expect hot loops to be mostly *type-stable*, meaning that the types of values are invariant. (12) For example, we would expect loop counters that start as integers to remain integers for all iterations. When both of these expectations hold, a trace-based compiler can cover the program execution with a small number of type-specialized, efficiently compiled traces.

Each compiled trace covers one path through the program with one mapping of values to types. When the VM executes a compiled trace, it cannot guarantee that the same path will be followed or that the same types will occur in subsequent loop iterations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06...$5.00

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

section, a user can choose 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 the 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.

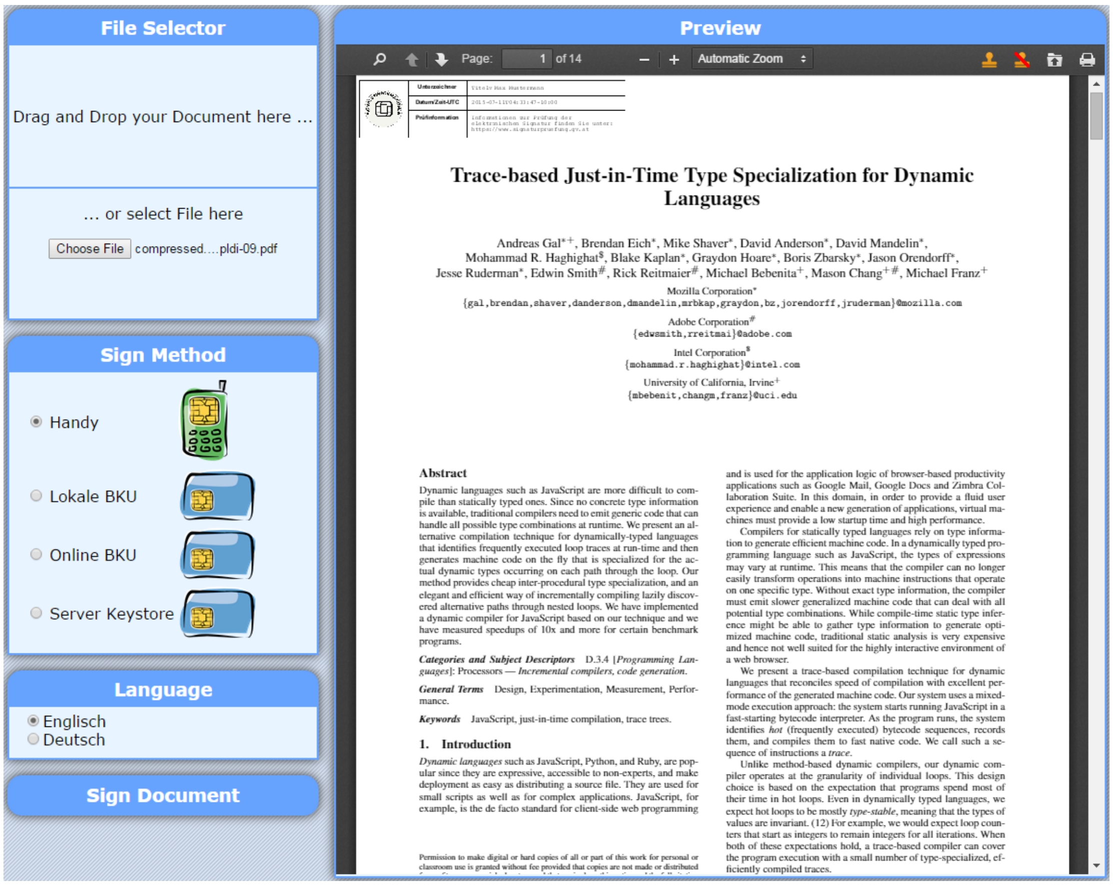
A main component of the website is the *Preview* section. Within that, a user gets a visual representation of the PDF document which he has previously selected. He can scroll through the different pages of the document, adjust the zoom, open a different file, print the file, place a signature etc. .

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 Mobile BKU the user can switch between the different sign methods directly within the *PDF.js* instance and with the button, the



Sign Method: [✓ Handy] Lokale BKU Online BKU Server Keystore

Language: [✓ Englisch] Deutsch

Unterzeichner	Titelv Max Mustermann
Datum/Zeit-UTC	2015-07-11T04:33:47-10:00
Prüfinformation	Informationen zur Prüfung der elektronischen Signatur finden Sie unter: https://www.signaturpruefung.gv.at

Trace-based Just-in-Time Type Specialization for Dynamic Languages

Andreas Gal[*+], Brendan Eich[*], Mike Shaver[*], David Anderson[*], David Mandelin[*],
Mohammad R. Haghighat[$], Blake Kaplan[*], Graydon Hoare[*], Boris Zbarsky[*], Jason Orendorff[*],
Jesse Ruderman[*], Edwin Smith[#], Rick Reitmaier[#], Michael Bebenita[+], Mason Chang[+#], Michael Franz[+]

Mozilla Corporation[*]
{gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Adobe Corporation[#]
{edwsmith,rreitmai}@adobe.com

Intel Corporation[$]
{mohammad.r.haghighat}@intel.com

University of California, Irvine[+]
{mbebenit,changm,franz}@uci.edu

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of 10x and more for certain benchmark programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — *Incremental compilers, code generation.*

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, just-in-time compilation, trace trees.

1. Introduction

Dynamic languages such as JavaScript, Python, and Ruby, are popular since they are expressive, accessible to non-experts, and make deployment as easy as distributing a source file. They are used for small scripts as well as for complex applications. JavaScript, for example, is the de facto standard for client-side web programming and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code, traditional static analysis is very expensive and hence not well suited for the highly interactive environment of a web browser.

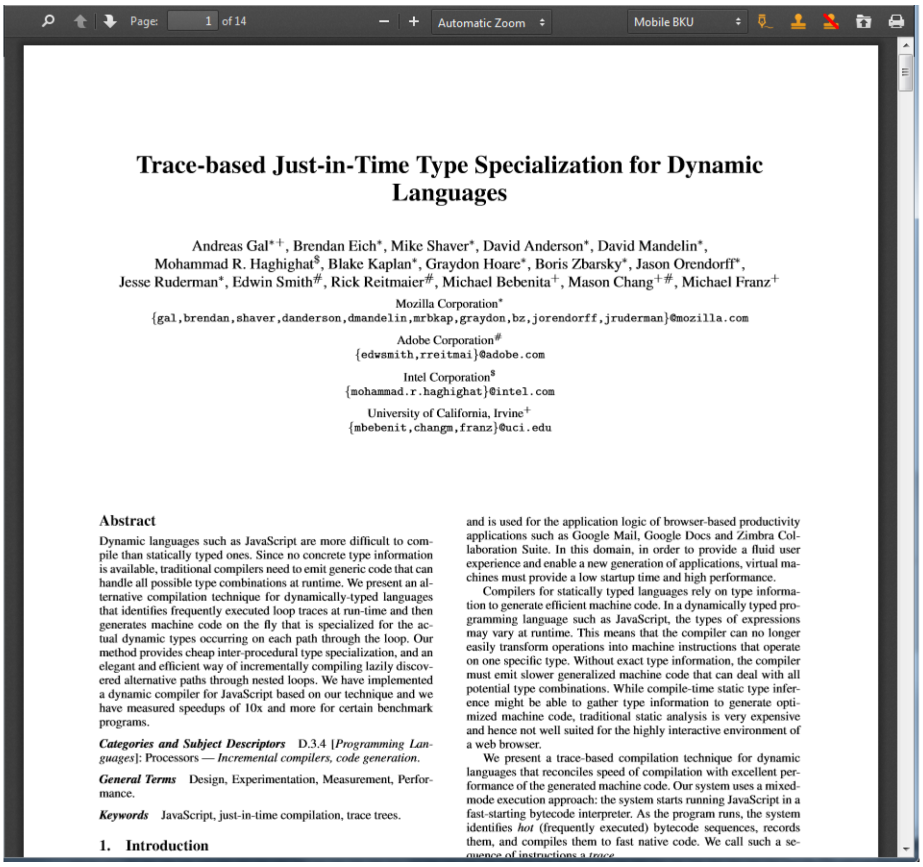
We present a trace-based compilation technique for dynamic languages that reconciles speed of compilation with excellent performance of the generated machine code. Our system uses a mixed-mode execution approach: the system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies *hot* (frequently executed) bytecode sequences, records them, and compiles them to fast native code. We call such a sequence of instructions a *trace*.

Unlike method-based dynamic compilers, our dynamic compiler operates at the granularity of individual loops. This design choice is based on the expectation that programs spend most of their time in hot loops. Even in dynamically typed languages, we expect hot loops to be mostly *type-stable*, meaning that the types of values are invariant. (12) For example, we would expect loop counters that start as integers to remain integers for all iterations. When both of these expectations hold, a trace-based compiler can cover the program execution with a small number of type-specialized, efficiently compiled traces.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed

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



Page: 1 of 14 Automatic Zoom Mobile BKU

Trace-based Just-in-Time Type Specialization for Dynamic Languages

Andreas Gal*+, Brendan Eich*, Mike Shaver*, David Anderson*, David Mandelin*, Mohammad R. Haghighat$, Blake Kaplan*, Graydon Hoare*, Boris Zbarsky*, Jason Orendorff*, Jesse Ruderman*, Edwin Smith#, Rick Reitmaier#, Michael Bebenita+, Mason Chang+#, Michael Franz+

Mozilla Corporation*
{gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Adobe Corporation#
{edwsmith,rreitmai}@adobe.com

Intel Corporation$
{mohammad.r.haghighat}@intel.com

University of California, Irvine+
{mbebenit,changm,franz}@uci.edu

Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of 10x and more for certain benchmark programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — *Incremental compilers, code generation.*

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, just-in-time compilation, trace trees.

1. Introduction

and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all potential type combinations. While compile-time static type inference might be able to gather type information to generate optimized machine code, traditional static analysis is very expensive and hence not well suited for the highly interactive environment of a web browser.

We present a trace-based compilation technique for dynamic languages that reconciles speed of compilation with excellent performance of the generated machine code. Our system uses a mixed-mode execution approach: the system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies *hot* (frequently executed) bytecode sequences, records them, and compiles them to fast native code. We call such a sequence of instructions a *trace*

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.

```
function webViewerLoad(evt) {
  //PDFViewerApplication.initialize().then(webViewerInitialized);
  PDFViewerApplication.initialize().then(function() {
    webViewerInitialized();
    displayPdf()
  });
}
```

State: webViewerLoad after the change

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

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 dragNdrop.js and app.js works).

```
window.addEventListener('change', function webViewerChange(evt) {
  ...
  var file = files[0];

  //ADDED
  setFile(file);

  if (!PDFJS.disableCreateObjectURL &&
  ...
}
```

State: webViewerChange after the change

```
window.addEventListener('change', function webViewerChange(evt) {
  ...
  var file = files[0];

  if (!PDFJS.disableCreateObjectURL &&
  ...
}
```

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.

```
function webViewerInitialized() {
  ...
  /*if (file && file.lastIndexOf('file:', 0) === 0) {
    // file:-scheme. Load the contents in the main thread because QtWebKit
    // cannot load file:-URLs in a Web Worker. file:-URLs are usually loaded
    // very quickly, so there is no need to set up progress event listeners.
    PDFViewerApplication.setTitleUsingUrl(file);
    var xhr = new XMLHttpRequest();
    xhr.onload = function() {
      PDFViewerApplication.open(new Uint8Array(xhr.response), 0);
    };
    try {
      xhr.open('GET', file);
      xhr.responseType = 'arraybuffer';
      xhr.send();
    } catch (e) {
      PDFViewerApplication.error(mozL10n.get('loading_error', null,
        'An error occurred while loading the PDF.'), e);
    }
    return;
  }

  if (file) {
    PDFViewerApplication.open(file, 0);
  }*/
}
```

State: webViewerInitialized after the change

```
function webViewerInitialized() {
  ...
  if (file && file.lastIndexOf('file:', 0) === 0) {
    // file:-scheme. Load the contents in the main thread because QtWebKit
    // cannot load file:-URLs in a Web Worker. file:-URLs are usually loaded
    // very quickly, so there is no need to set up progress event listeners.
    PDFViewerApplication.setTitleUsingUrl(file);
    var xhr = new XMLHttpRequest();
    xhr.onload = function() {
      PDFViewerApplication.open(new Uint8Array(xhr.response), 0);
    };
    try {
      xhr.open('GET', file);
      xhr.responseType = 'arraybuffer';
      xhr.send();
    } catch (e) {
      PDFViewerApplication.error(mozL10n.get('loading_error', null,
        'An error occurred while loading the PDF.'), e);
    }
    return;
  }

  if (file) {
    PDFViewerApplication.open(file, 0);
  }
}
```

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

```
function placeSignature(evt) {
  ...
}
```

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 placed on is extracted from *PDF.js*

```
function makeSignatureDraggable(signature) {
  ...
}
```

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

```
function updateSignaturePosition(signature) {
  ...
}
```

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.

```
function removeSignature() {
  ...
}
```

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: Valules 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

```
function sign(statusObj) {
  var file = statusObj.getFile();
  if(!file) {
    alert("No File Opened");
    return;
  }
  var fd = new FormData();
  fd.append("source", "internal");
  fd.append("connector", global_status.getConnector());
  fd.append("pdf-file", global_status.getFile());

  if(isSignaturePlaced()) {
    fd.append("sig-pos-x", global_status.getSignature().posx);
    fd.append("sig-pos-y", global_status.getSignature().posy);
    fd.append("sig-pos-p", global_status.getSignature().page);
  }

  $.ajax({
    url: "/pdf-as-web/Sign",
    data: fd,
    processData: false,
    contentType: false,
    type: "POST",
    success: function(response) {
      $("#fade").remove();
      $("#popup").remove();
      var fade_div = "<div id='fade' class='black_overlay'></div>";
      var popup_div = "<div id='popup' class='white_content'><a href='javascript
          :void(0)' id='closelink'>Close</a><div id='resp'></div></div>"
      $("body").append(fade_div);
      $("body").append(popup_div);
      $("#resp").html(response);
      $("#closelink").bind("click", function(evt) {
        $("#fade").remove();
        $("#popup").remove();
      });
    }
  });
}
```

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.