Electronic Documents

`execJS`: A new technique for introducing discardable JavaScript into a PDF file from a LaTeX source

D. P. Story

1 Introduction

This article describes a technique, referred to as `execJS`[1], for writing JavaScript from within a LaTeX source file that will be executed once when the document is first opened, then *discarded*. The `execJS` technique allows you to execute JavaScript methods, *even ones that have their use restricted for security reasons*. The discardable code and the ability to execute even security-restricted methods are the distinguishing features of `execJS`.

The method requires Acrobat 5.0 or later (the full application, not the Acrobat Reader), *or* Acrobat Approval 5.0 or later. You can, though not required, use the Acrobat Distiller to create the PDF document; or, as so many do, you can use either the `pdftex` or `dvipdfm` applications. Once the PDF document is completed, it can be viewed by the Acrobat Reader 4.0 or higher.[2]

The technique is meant to be used in document development, preparation and assembly for authors who want to tap into the power methods of JavaScript.

[1] execute JavaScript

2 Summary of the technique

The `execJS` technique consists to two parts:

1. A new LaTeX environment, the `execJS` environment
2. A few lines of folder-level JavaScript

The `execJS` environment is implemented as part of the insdljs Package[3], which was written in preparation for the TeX Users Group 2001 Conference.

Within the `execJS` environment the "discardable" JavaScript is written to an auxiliary file with an extension of `.fdf`. When the newly created PDF document is opened for the first time in the viewer (either Acrobat or Approval) the `.fdf` file is imported into the document, and the JavaScript contained within the file is executed.

The second part of the technique, the folder-level JavaScript, gives the "discardable" JavaScript the right to execute security-restricted JavaScript methods.

3 Animated motivation

In the past few years, document authors who are producing interactive PDF documents from a LaTeX source have grown significantly in number.[4] Many authors, myself included, crave to have the ability to use Acrobat's powerful JavaScript interpreter to its fullest, all from a LaTeX source. The development of the `execJS` technique is a significant step in that direction. The `execJS` method came about by my pursuit of a holy grail of LaTeX/PDF production, *animation*. One of the examples that appears in this article is a PDF animation created entirely within a LaTeX source file!

4 The `execJS` technique

Acrobat Version 5.0 comes with an extended FDF (Forms Data Format) specification. This specification creates a new `Doc` key, the value of which refers to JavaScript contained within the FDF file that is to

[2] Version 5.0 or later of Reader is required if JavaScript objects, properties or methods are used not available in version 4.0.

[3] The insdljs Package, a standalone package, is distributed as a component package of the AcroTeX eDucation Bundle: `http://www.math.uakron.edu/~dpstory/webeq.html`

[4] For links to and descriptions of some of the many authors doing quality work on the Web, see the AcroTeX web site: `http://www.math.uakron.edu/~dpstory/acrotex.html`

be imported into the document as Document Level JavaScript (DLJS). See the paper (Story, 2001) for details of how to use this specification to insert DLJS from a LaTeX source. DLJS can be inserted from your LaTeX source file using the insdljs Package. The specification also defines an `After` key, the value of which is an indirect reference to JavaScript code also contained in the FDF file.

The `After` key is the one of interest in this paper. The JavaScript referenced by the `After` key is executed after the FDF is imported into the document. The JavaScript is executed *but not saved*, it is "discardable".

Consider the following FDF file containing both the `Doc` and `After` keys.

```
%FDF-1.2
1 0 obj
<< /FDF
    << /JavaScript
        << /Doc 2 0 R /After 3 0 R
        >>
    >>
>>
endobj
2 0 obj
[ (ExecJS execjs) (var _execjs = true;) ]
endobj
3 0 obj
<<>>
stream
app.alert("\"Discardable\" code executed!");
endstream
endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

When this file is imported into Acrobat (or Approval), one line of code will be placed at the document level, `var _execjs = true`, and one line of code

```
app.alert("\"Discardable\" code executed!");
```

will be executed, but not saved.

4.1 The execJS environment

It was a simple modification of the insdljs Package to implement an `execJS` environment that writes a FDF file containing the `After` key. Thus,

```
\usepackage[execJS]{insdljs}
...
\begin{execJS}{execjs}
app.alert("\"Discardable\" code executed!");
\end{execJS}
```

writes the file seen in the FDF above. The required argument for this environment is the base file name of the FDF to be written.

The environment not only writes the FDF file, if the `execJS` option of the `insdljs` Package is specified, it also adds an open page action to the first page of the document:

```
if (typeof _execjs == "undefined")
    this.importAnFDF("execjs.fdf");
```

This is a point in the whole technique where Acrobat or Approval is required. The JavaScript method `this.importAnFDF()` is not available for the Acrobat Reader.

When the document is opened for the first time, the file `execjs.fdf` gets imported into the PDF document. The FDF also inserts the DLJS `var _execjs = true` (which will be saved with the document). This variable declaration prevents the FDF from being repeatedly imported each time the first page is viewed.

4.2 Folder JavaScript

The `execJS` environment writes the FDF file which, in turn, is imported into the newly created PDF, the JavaScript is executed, and is "discarded" (not saved); however, security-restricted JavaScript still cannot be executed. An additional trick is needed.

Many of the restricted methods can be used only during, what Acrobat calls, menu, console or batch events. The approach taken here is to execute JavaScript through a menu event. To do this, create a file, called `myJS.js` (or any name with an extension of `js`). The contents of this file are the lines:

```
_MenuProc = function() {;}
app.addMenuItem({
    cName: "MenuProc",
    cUser: "Menu Procedure",
    cParent: "Tools",
    cExec: "_MenuProc()",
    nPos: 0
});
```

The first line defines _MenuProc, a JavaScript function which does nothing. The rest of the lines define a new menu item under the "Tools" menu. Additional menu items can only be created through a folder level JavaScript file.

Place `myJS.js` in the `JavaScripts` folder, follow the path `Acrobat 5.0/Acrobat/JavaScripts`. This is the folder in which `aform.js`, the folder level JavaScript for the forms plugin, is kept.

The trick for executing JavaScript methods that are restricted to a menu event is to redefine the Folder level function _MenuProc. Consider the following code withing the `execJS` environment:

```
\begin{execJS}{execjs}
function importMyIcons ()
{
   for ( var i=0; i < 36; i++)
     this.importIcon("rotate"+i,"animation.pdf",i);
}
_MenuProc = importMyIcons;
app.execMenuItem("MenuProc");
_MenuProc = function() {;}
\end{execJS}
```

A function `importMyIcons` is defined that imports a series of named icons using the `this.importIcon()`. The use of this method is restricted to menu, console or batch events if a file name is given as the second argument. The first argument is the name of the icon newly embedded in the document.

Following the function definition, we make the assignment

`_MenuProc = importMyIcons;`

then execute the menu item named `"MenuProc"`, which, in turn, causes `_MenuProc` (now assigned as `importMyIcons`) to be executed. (Multiple function definitions and assignments can be made in this way.) Finally, the `_MenuProc` function is reassigned its default definition (optional).

Important: After the document is assembled, save the document (as you can with Acrobat and Approval), this will save the DLJS, but not the "discardable" code.

See the Acrobat JavaScript Object Specification, Version 5.0 or later,[5] for details of the JavaScript methods just illustrated.

5 An animation example

In this section, an animation example is presented. The animation is done completely within the LaTeX source file. After the PDF document is finally assembled, the animation is ready to run.

The animation was done using two LaTeX files, `execjstst.tex` and `animation.tex`. The AcroTeX eDucation Bundle[6] was used. The AcroTeX Bundle includes the following packages:

- The Web Package: For creating good looking PDF documents for the Web.
- The Exerquiz Package: For creating online exercises and quizzes. (In the animation, the form field macros only were used.) Exerquiz loads the insdljs Package and passes the options of insdljs to it.
- The insdljs Package: A package used to insert Document Level JavaScripts, to create open actions, and to create immediatelly executable and "discardable" JavaScript.
- The dljslib Package: A library of useful JavaScript functions that can be "checked out" for use.

[5] Available under the `Help` menu of Acrobat, or at `http://partners.adobe.com/asn/developer/technotes/acrobatpdf.html`

[6] Available at CTAN:`/tex-archive/macros/latex/contrib/supported/webeq` or at the Bundle's homepage `http://www.math.uakron.edu/~dpstory/webeq.html`

5.1 animation.tex

The `animation.tex` file is the LaTeX source used to create the series of images that will be display in the animation. This animation is rather simple. PSTricks was used to create a series of 36 graphics:

```
\documentclass{article}
\usepackage{pstricks,pst-plot}
\usepackage[dvipsone]{web}%<- or dvips

\margins{0pt}{0pt}{0pt}{0pt} % no margins
\screensize{1in}{1in}        % 1in by 1in

\pagestyle{empty}\parindent=0pt
\SpecialCoor

\begin{document}
\psset{unit=1in,origin={-.5,-.5}}

\multido{\i=90+-10}{36}{%
    \begin{pspicture}(1in, 1in)
        \psframe[fillstyle=solid,
            fillcolor=lightgray,
            linecolor=lightgray](-.5,-.5)(.5,.5)
        \psline[linecolor=blue](0,0)(.5; \i)
        \pscircle[linecolor=red](0,0){.5}
    \end{pspicture}
    \newpage
}
\end{document}
```

This particular file was distilled to create the PDF document, `animation.pdf`. For more sophisticated animations, the graphic images need to be created using some high-end application.

5.2 ExecJStst.tex

The file `execjstst.tex` is actually the animation demo. Here is the verbatim listing of this file.

```
\documentclass{article}
\usepackage
    [dvipsone, % <- dvips, pdftex, dvipdfm
     designi,
    ]{web}
% exerquiz loads insdljs, and passes execJS to it
\usepackage[execJS]{exerquiz}

% Embed the animation images as named icons, the ith
% icon is named rotatei, i=0..35
\begin{execJS}{execjs}
function importMyIcons ()
{
  for ( var i=0; i < 36; i++)
      this.importIcon("rotate"+i,"animation.pdf",i);
}
_MenuProc = importMyIcons;
app.execMenuItem("MenuProc");
_MenuProc = function() {;}
\end{execJS}

% Define a JavaScript action that will be attached
% to the button "aniCtrl" that starts the
% animation.
\newcommand{\aniCtrlAction}
{%
  /A << /S /JavaScript /JS
  (%
    function ShowIt()\jsR
    {\jsR\jsT
      var oIcon;\jsR\jsT
      oIcon=this.getIcon
                ("rotate"+run.count);\jsR\jsT
      f.buttonSetIcon( oIcon, 0);\jsR\jsT
      run.count++;\jsR\jsT
      run.count \%= 36;\jsR
    }\jsR
    var f = this.getField("myAnimation");\jsR
    var run = app.setInterval("ShowIt()",100);\jsR
    run.count = 0;\jsR
    var timeout = app.setTimeOut
           ("app.clearInterval(run);", 3*3600+200);
  ) >>
}

\begin{document}
\section*{An Animation}

% Center the animation "window" and the start button
\begin{center}
    \eqIcon{myAnimation}{72bp}{72bp}\\
    \eqGenButton[\CA{Push}\rawPDF{\aniCtrlAction}]
       {aniCtrl}{36bp}{16bp}
\end{center}
\end{document}
```

Many of the above commands come from the exerquiz Package. The macros `\jsR` and `\jsT` expand to `\r` and `\t`, which are escape sequences in JavaScript for newline and tab; `\eqIcon` makes it easy to create a button field set up for displaying a (PDF) icon as its face appearance; `\eqGenButton` is a general command for creating push buttons.

The JavaScript action defined by LaTeX command, `\aniCtrlAction`, first defines a JavaScript function called `ShowIt`, which gets the *icon object* using the `getIcon` method, sets the button appearance of the `myAnimation` field using the field method `buttonSetIcon` method, and increments a counter; at the top-level, the script gets the *field object* for the `myAnimation` field, starts a timing event which calls the function `ShowIt` every 100 milliseconds, initializes the counter, and sets the timeout interval.

6 Another example

Recently, I produced a PDF version of an `html` online survey for the **Seybold PDF Conference 2002**. The online version was an "intelligent" survey: the answer to one question determined what question would next be posed to the respondent.

LaTeX and the AcroTeX Bundle were used to duplicate the questions and the functionality of the `html` version in PDF. A short macro package was written, DLJS was automatically introduced into the document using the insdljs package which supported the "logic" of the survey.

The PDF version uses form templates. (A form template is a special type of page that can be hidden or made visible, a copy of a template can be spawned, Acrobat viewer or Acrobat Approval required.) Templates were used to reveal the next page, based on the answers given on the current page by the respondent.

Templates cannot be created using the `pdfmark` operator (`pdftex` and `dvipdfm` do not support their creation), but can be created using certain security-restricted JavaScript methods. The `execJS` technique was used to create the templates. As a result, a fairly complex PDF document was assembled entirely from the content and commands in the LaTeX source file.

Read the article "Seybold PDF survey in PDF also worthy of study"[7] by Kurt Foss, Planet PDF Editor[8], for more details of the operational capability of the PDF version of the survey.

7 In conclusion

LaTeX has become a powerful markup language for creating interactive PDF documents; in my opinion, the LaTeX system is *the premier* system for building, debugging, modifying and assembling complex PDF documents. The hyperref and exerquiz packages can be used to create links and form fields with JavaScript actions attached, the insdljs package allows for the insertion of document level JavaScript into the PDF document, open actions, and now executable, "discardable" JavaScript that can be used in a post-creation setting.

References

Story, D. P. "Techniques of introducing document-level JavaScript into a PDF file from a LaTeX source". *TUGboat* **22**(3), 161–167, 2001.

◇ D. P. Story
Department of Theoretical and Applied Mathematics
The University of Akron
Akron, OH 44278
dpstory@uakron.edu
http://www.math.uakron.edu/~dpstory/

[7] www.planetpdf.com/mainpage.asp?webpageid=2130
[8] Planet PDF: www.planetpdf.com