pdfHTML

Convert HTML/CSS to PDF

www.itextpdf.com

Due to the popularity of the World Wide Web, its underlying technologies have become commonplace and ubiquitous. HTML (and CSS) in particular are widely used to present text and information in a visually pleasing manner. At the same time, many companies rely on the PDF format for document workflows, internal documentation and archiving. The main draws of PDF are the reliability and portability of the format, and the commitment to present the same appearance across different devices, which is an explicit mission statement of the PDF specification.

PDF however, is not an easily editable format, and is often converted from other formats as a final step in document creation. One such format can be HTML, pdfHTML leverages the widespread knowledge of the format and existing skills of development resources in converting HTML to PDF. pdfHTML provides the engine that converts HTML/CSS content to a well-formatted, well-structured PDF document, without the need to know the technical details of the PDF format.

HTML/CSS to PDF transformation

In HTML, content is wrapped in HTML tags. Each tag corresponds to a conceptual element (e.g. paragraph, table), and many tags can be nested. These tags can be organized into a hierarchical model, such as the DOM (Document Object Model) of the HTML document, detailing the structure and semantics of the content.

Styling and visual representation for HTML content is provided by the use of Cascading Style Sheets (CSS). CSS declarations define styling and layout information (e.g. font, font-size, margins, borders, color, alignment, etc.) for the various HTML tags and their content. These can be found in the HTML file itself, or as a separate style-sheet file.

A web browser parses and interprets the HTML file and accompanying CSS to create a visual representation of the content, calculating the rendering and layout on the fly and visualizing the various contents according to their CSS declarations and the renderers own settings.

In contrast, at its core, a PDF document is not inherently structured and semantic. Its content consists of a set of instructions that result in painting at absolute positions on a large canvas. The concept of a line of text, for example, does not exist at this basic level. We only infer that visually because characters appear next to each other at the same vertical position.

PDF does offer an additional layer of functionality to store semantic and structural information, using similar concepts as HTML: tagging pieces of content according to their roles in the document and constructing a hierarchical tree. To support features like proper content extraction, repurposing of content, search indexing and accessibility, it's crucial to augment the visual-only representation of PDF documents with this additional information.

Since HTML documents inherently contain semantic and structural information, they are an excellent source to convert to rich, smart PDF documents. This is where pdfHTML comes in.

What is pdfHTML?

pdfHTML is an add-on module for the iText7 platform that transforms HTML and accompanying CSS into a PDF document. Built for the creation of PDF files from HTML templates, pdfHTML allows you to automate PDF generation for documents like internal reports, tickets, invoices, and more.

pdfHTML provides you with the means to create beautiful and functional PDFs without forcing designers to learn the complex PDF syntax or the intricacies of the iText 7 platform. They simply need to utilize their common, well-honed HTML and CSS skills to create the template, and pdfHTML takes care of the rest, converting to a visually equivalent PDF document, with the possibility to preserve semantic, structural and accessibility information.

How does it work?

A technical in-depth look

On a conceptual level, pdfHTML works by mapping HTML tags to iText 7 layout objects, and CSS property declarations to iText layout properties. On a practical level, this process happens through the use of `TagWorkers` and `CssAppliers`.

What's a TagWorker?

A `TagWorker` is a class responsible for processing an HTML tag. This processing includes resolving any resources required by the tag and its content; translating the tags' content into an iText layout element; resolving any non-style attribute and resolving any style attributes through a `CssApplier`.

What's a CssApplier?

A `CssApplier` is a class responsible for the processing of CSS styles and any declarations for a HTML tag. The implementation of this `CssApplier` contains all the necessary logic to resolve and apply the style declarations to the iText layout element that is associated with the `CssApplier`.

Each HTML tag is mapped to a `TagWorker` and `CssApplier`, and those classes contain the necessary logic to process the tag, selecting the iText layout object it corresponds to and applying any necessary CSS. When processing the HTML DOM, pdfHTML walks through the tree in a depth-first manner, starting the translation on a tag, and then recursively processing all its children, ending the processing when all its children have been processed. The visualization of this process can be seen in figures 1 and 2:

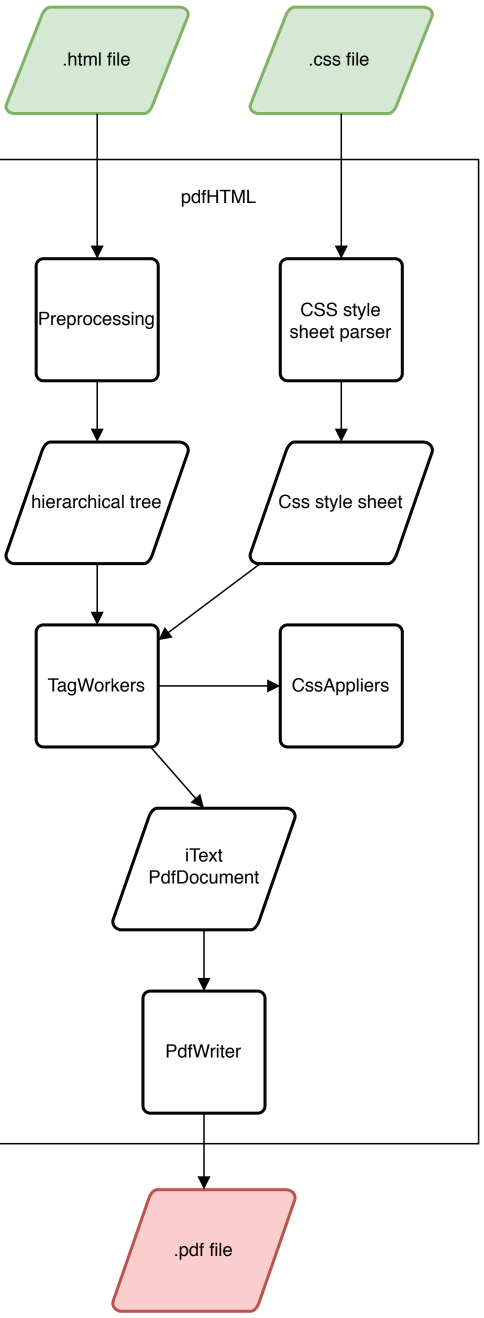


Figure 1: pdfHTML high level flow

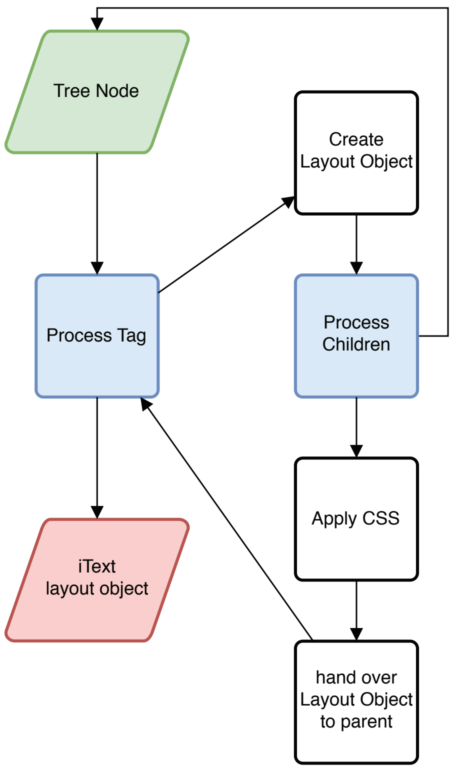


Figure 2: pdfHTML tag processing

In practice

A basic example will show the use of pdfHTML. For this, we will use the following HTML and CSS.

```
<html>
     <head>
            <link rel="stylesheet" type="text/css" href="simple.css"/>
     </head>
     <body>
            <p>iText pdfHTML</p>
            <div>Converting HTML to PDF with (Cascading) Style (Sheets)!</div>
     </body>
</html>
```

HTML code

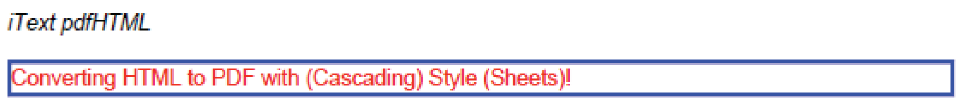
```
p{
     font-style:italic;
}
div{
     color:red;
     border-style:solid;
     border-width:2pt;
     border-color:blue;
}
```

simple.css

The output will be directly written to a PDF file, using the following code:

```
ConverterProperties converterProperties = new ConverterProperties();
HtmlConverter.convertToPdf(new FileInputStream(htmlSource), new FileOutputStream(pdfDest),
converterProperties);
```

The resulting PDF will look as follows:


iText pdfHTML

Converting HTML to PDF with (Cascading) Style (Sheets)!

Figure 3: pdfHTML output

Let us explain what happened, in order:

1. The HTML file is transformed into a DOM representation of the content.
2. All CSS declarations are extracted from both the HTML as well as the style sheet, and are resolved.
3. The html tag is processed by the `HTMLTagWorker`, which creates a layout document object.
4. The head and body tags are special cases and do not have a corresponding `TagWorker`. They're skipped in the walkthrough.
5. The p tag is encountered and a `PTagWorker` creates a Paragraph object. The text iText pdfHTML is encountered next, and added to the paragraph created in its parent. A `BlockCssApplier` is created for the p tag , and applies the italic to the text. The `PTagWorker` finished its work and the paragraph is added to the document.
6. The div tag is encountered and a `DivTagWorker` creates a Div object. The text Converting HTML to PDF with (Cascading) Style (Sheets)! is encountered next, and added to the div created by the `DivTagWorker`. A `BlockCssApplier` is created for the div tag, and it applies the red colour to the text, and the blue border surrounding it to the Div layout object.
7. The `DivTagWorker` finishes its work and the Div is added to the document.

The `HTMLTagworker` finishes its work and passes the document back to the `HTMLConverter`, which closes it, writing the contents to the output stream.

Some more in-depth information

Out of the box, pdfHTML offers a range of additional configuration options. No extra knowledge of PDF is required.

Configuration settings can be enabled with the optional last parameter `ConverterProperties` of all `HTMLConverter` public methods. This parameter contains the basic configuration options that allow users to customize handling of the input data in various ways. The following options allow the configuration of your pdfHTML code for optimal results.

baseUri

If the HTML file requires any external resources, such as a standalone CSS file or image files, pdfHTML will need to know where these elements are located. That location may either be a URI on the local file system or online.

pdfHTML will try to use a reasonable default value in case you don't specify a baseUri. If you use a String parameter to pass your HTML, then the default value will be the folder where the code is executed. If you use the overload of convertToPdf with a File parameter, it will use the same location as the input file.

If neither of these defaults is applicable, you will need to define the default resource location root. Any references in the HTML file will start from that path in order to find the resource. It is possible to use the ../ syntax to go up in the directory tree of your file system, but for most purposes, the specified URI acts as the root folder for determining paths. Thus e.g. `<img src="static/img/logo.png"/>` and `<img src="/static/img/logo.png"/>` will both refer to the same file, relative to the baseUri folder you specified or to the default value.

fontProvider

To customize the fonts for use by pdfHTML, you can define a `FontProvider` object that will act as a repository for those fonts.

There is a standard implementation available in the pdfHTML project which is called `DefaultFontProvider`. You can specifically determine which fonts should be supplied to the PDF rendering logic. On top of the basic behavior that you can define in the constructor, you can also add a font that you specify as a location on the system, a byte[], or an iText `FontProgram` object.

`DefaultFontProvider` has a constructor with three boolean arguments, which allow you to specify (in order):

- Whether or not to select the 14 standard PDF fonts
- Whether or not to select a number of Free fonts included with pdfHTML
- Whether or not to try and select all fonts installed in the system's default font folder(s)

There is also a default no-argument constructor for `DefaultFontProvider`. This constructor will select the 14 standards fonts as well as the fonts shipped with iText and pdfHTML but will not register the system fonts by default.

In addition to this, fonts can be added by either supplying the path to a font file or by telling the `FontProvider` to register all fonts contained in a specified folder.

```
ConverterProperties props = new ConverterProperties();
FontProvider dfp = new DefaultFontProvider(true, false, false);
dfp.addFont("/path/to/MyFont.ttf");
props.setFontProvider(dfp);
```

mediaDeviceDescription

If the input file uses media queries, you can simply tell pdfHTML to interpret the relevant set of rules:

```
ConverterProperties props = new ConverterProperties();
props.setMediaDeviceDescription(new MediaDeviceDescription(MediaType.PRINT));
```

The registered MediaType constants are:

- MediaType.ALL
- MediaType.AURAL
- MediaType.BRAILLE
- MediaType.EMBOSSED
- MediaType.HANDHELD
- MediaType.PRINT
- MediaType.PROJECTION
- MediaType.SCREEN
- MediaType.SPEECH
- MediaType.TTY
- MediaType.TV

A use case: Accessible PDF creation

In a company, policies can impose strict guidelines concerning aspects of documents, as e.g. the accessibility of documents.

With minimal effort, pdfHTML can be configured to generate PDFs that comply with the accessibility standards as defined in PDF/A and PDF/UA, by leveraging the functionality provided by the iText 7 core engine regarding the setting of both the tagging as well as adding the additional accessibility information.

```
public void createPdf(String src, String dest, String resources) throws IOException {
     try {
            FileOutputStream outputStream = new FileOutputStream(dest);
            writerProperties writerProperties = new WriterProperties();
            writerProperties.addXmpMetadata();

            PdfWriter pdfWriter = new PdfWriter(outputStream, writerProperties);
            PdfDocument pdfDoc = new PdfDocument(pdfWriter);
            pdfDoc.getCatalog().setLang(new PdfString("en-US"));

//Set the pdf to be tagged
            pdfDoc.setTagged();
            pdfDoc.getCatalog().setViewerPreferences(new PdfViewerPreferences().
            setDisplayDocTitle(true));

//Set meta tags

            pdfDocumentInfo pdfMetaData = pdfDoc.getDocumentInfo();
            pdfMetaData.setAuthor("Samuel Huylebroeck");
            pdfMetaData.addCreationDate();
            pdfMetaData.getProducer();
            pdfMetaData.setCreator("iText Software");
            pdfMetaData.setKeywords("HTML, PDF");
            pdfMetaData.setSubject("PDF accessibility");
//Title is derived from html

//Setup custom header role, mapping it to heading

            pdfDoc.getStructTreeRoot().getRoleMap().put(new PdfName("Heading"),PdfName.H);

// pdf conversion

            ConverterProperties props = new ConverterProperties();
            FontProvider fp = new FontProvider();
            fp.addSystemFonts();
            props.setFontProvider(fp);
            props.setBaseUri(resources);

//Setup custom tagworker factory for better tagging of headers

            DefaultTagWorkerFactory tagWorkerFactory = new AccessibilityTagWorkerFactory();
            props.setTagWorkerFactory(tagWorkerFactory);
            HtmlConverter.convertToPdf(new FileInputStream(src), pdfDoc, props);
            pdfDoc.close();

      } catch (Exception e) {
            e.printStackTrace();
      }

}
```

How to customize pdfHTML

In addition to the out-of-the-box functionality of pdfHTML, based on the capabilities of the iText7 platform, and its configuration options, pdfHTML also enables additional customization opportunities, as discussed in this section.

There are two plugin mechanisms in pdfHTML that allow you to execute custom behavior in your HTML/CSS conversion process. Both have a very similar working.

Much like the configuration options as specified in the associated pdfHTML white paper, registering your plugin code with pdfHTML happens through the `ConverterProperties`:

```
ConverterProperties props = new ConverterProperties();
props.setTagWorkerFactory(new MyTagWorkerFactory()); // Custom HTML parsing
props.setCssApplierFactory(new MyCssApplierFactory()); // Custom CSS parsing
```

TagWorkerFactory

To define custom rules for existing HTML tags, you can create a `ITagWorker` implementation that will execute logic defined by you. The most common use cases are handling a tag in a nonstandard way or as a no-op, but you can also implement a custom tag for your specific purposes. After implementing this interface or extending an existing implementation, you still need to register it with pdfHTML so that it knows what to call.

This can be achieved by extending `DefaultTagWorkerFactory` and overriding the following method:

```
public class MyTagWorkerFactory extends DefaultTagWorkerFactory {
     public ITagWorker getCustomTagWorker(IElementNode tag, ProcessorContext context) {
            if (tag.name().equalsIgnoreCase("custom")) {
                   return new CustomTagWorker(); // implements ITagWorker
            }
            if (tag.name().equalsIgnoreCase("p")) {
                   return new CustomParagraphTagWorker(); // extends ParagraphTagWorker
            }
// default return value should be either null
// so the default implementations can be called ...
            return null;
// ... or you can directly call the superclass method
// for the exact same effect
            return super.getTagWorker(tag, context);
     }
}
```

One particular use case might be to add dynamic content to your PDF, such as barcodes. In that case, you can define `<qr>http://www.example.com</qr>` in the source HTML, rather than having to generate an image separately. Your custom `TagWorker` then leverages the iText APIs to create the QR code, and adds it to the document.

The input:

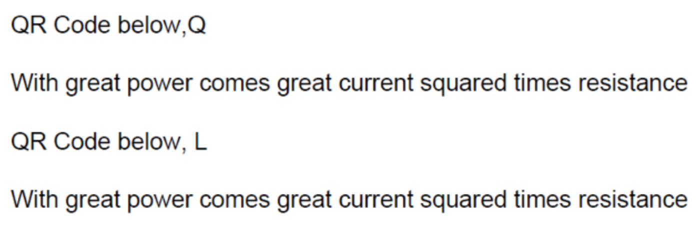
```
<!DOCTYPE html>

<html lang="en">
<head>
     <meta charset="UTF-8">
     <title>QRCode Example</title>
     <link rel="stylesheet" type="text/css" href="qrcode.css"/>
</head>
<body>
<p>QR Code below,Q </p>
<qr charset="Cp437" errorcorrection="Q">
With great power comes great current squared times resistance
</qr>
<p>QR Code below, L</p>
<qr charset="Cp437" errorcorrection="L">
     With great power comes great current squared times resistance
</qr>
</body>
</html>

qr{
     border:solid 1px red;
     height:200px;
     width:200px;

}
```

The output without a custom `TagWorker`:



QR Code below,Q

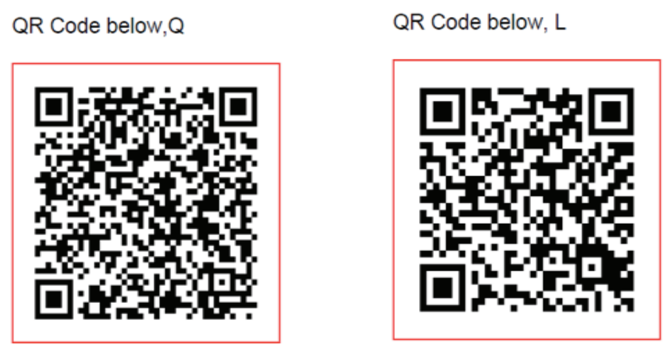
With great power comes great current squared times resistance

QR Code below, L

With great power comes great current squared times resistance

Figure 4: Output with no custom TagWorker

The output with a custom `TagWorker`:



QR Code below,Q

QR Code below, L

Figure 5: Output with Custom Tagworker

```java
import com.itextpdf.barcodes.BarcodeQRCode;
import com.itextpdf.barcodes.qrcode.EncodeHintType;
import com.itextpdf.barcodes.qrcode.ErrorCorrectionLevel;
import com.itextpdf.html2pdf.attach.ITagWorker;
import com.itextpdf.html2pdf.attach.ProcessorContext;
import com.itextpdf.html2pdf.html.node.IElementNode;
import com.itextpdf.layout.IPropertyContainer;
import com.itextpdf.layout.element.Image;

import java.util.HashMap;
import java.util.Map;

public class QRCodeTagWorker implements ITagWorker {
     private static String[] allowedErrorCorrection = {"L","M","Q","H"};
     private static String[] allowedCharset = {"Cp437","Shift_JIS","ISO-8859-
1","ISO-8859-16"};
     private BarcodeQRCode qrCode;
     private Image qrCodeAsImage;

     public QRCodeTagWorker(IElementNode element, ProcessorContext context){
//Retrieve all necessary properties to create the barcode
```

```
            Map<EncodeHintType, Object> hints = new HashMap<>();
//Character set
            String charset = element.getAttribute("charset");
            if(checkCharacterSet(charset )){
                   hints.put(EncodeHintType.CHARACTER_SET, charset);
            }
//Error-correction level
            String errorCorrection = element.getAttribute("errorcorrection");
            if(checkErrorCorrectionAllowed(errorCorrection)){
                   ErrorCorrectionLevel errorCorrectionLevel =
                   getErrorCorrectionLevel(errorCorrection);
                   hints.put(EncodeHintType.ERROR_CORRECTION, errorCorrectionLevel);
            }
//Create the QR-code
            qrCode = new BarcodeQRCode("placeholder",hints);

     }

     @Override
     public void processEnd(IElementNode element, ProcessorContext context) {
//Transform barcode into image
            qrCodeAsImage = new Image(qrCode.createFormXObject(context.getPdfDocument()));
     }
     @Override
     public boolean processContent(String content, ProcessorContext context) {
//Add content to the barcode
            qrCode.setCode(content);
            return true;
     }
     @Override
     public IPropertyContainer getElementResult() {
            return qrCodeAsImage;
     }
     @Override
     public boolean processTagChild(ITagWorker childTagWorker, ProcessorContext context) {
            return false;
     }

     private static boolean checkErrorCorrectionAllowed(String toCheck){
            for(int i = 0; i<allowedErrorCorrection.length;i++){
                   if(toCheck.toUpperCase().equals(allowedErrorCorrection[i])){
                          return true;
                   }
            }
            return false;
```

```
    }
    private static boolean checkCharacterSet(String toCheck){
        for(int i = 0; i<allowedCharset.length;i++){
            if(toCheck.equals(allowedCharset[i])){
                return true;
            }
        }
    return false;
    }
    private static ErrorCorrectionLevel getErrorCorrectionLevel(String level){
        switch(level) {
            case "L":
                return ErrorCorrectionLevel.L;
            case "M":
                return ErrorCorrectionLevel.M;
            case "Q":
                return ErrorCorrectionLevel.Q;
            case "H":
                return ErrorCorrectionLevel.H;
        }
        return null;
    }
}
```

The custom `tagWorkerFactory`

```
import com.itextpdf.html2pdf.attach.ITagWorker;
import com.itextpdf.html2pdf.attach.ProcessorContext;
import com.itextpdf.html2pdf.attach.impl.DefaultTagWorkerFactory;
import com.itextpdf.html2pdf.html.node.IElementNode;

public class QRCodeTagWorkerFactory extends DefaultTagWorkerFactory {

    @Override
    public ITagWorker getCustomTagWorker(IElementNode tag, ProcessorContext context) {
        if(tag.name().equals("qr")){
            return new QRCodeTagWorker(tag, context);
        }
        return super.getTagWorker(tag, context);
    }
}
```

cssApplierFactory

By default, pdfHTML will only execute CSS logic on standard HTML tags. If you have defined a custom HTML tag and you want to apply CSS to it, then you will also have to write an `ICssApplier` and register it by extending `DefaultTagWorkerFactory`.

Similarly, you may want to change the way a standard HTML tag reacts to a CSS property. In that case, you can extend the `ICssApplier` for that tag and write custom logic.

Integration with existing iText add-ons

pdfHTML can integrate with existing iText add-ons. In particular, it integrates seamlessly with pdfCalligraph - iText's advanced typography module. This add-on enables the correct representation of text in complex alphabets, such as Arabic and Hindi. Leveraging this module in an application requires absolutely no extra code: if the pdfCalligraph module is available on the classpath when the PDF instructions are written, its code will be used to create correctly expressed text in the relevant writing system.

This also holds true for pdfHTML: all you need is the JAR and a license file for pdfCalligraph, and your HTML parsing code will work out of the box.

Comparison to XMLWorker

iText’s previous HTML/CSS conversion tool is XML Worker in combination with iText 5. It has been replaced by pdfHTML in combination with iText 7. A quick comparison.

pdfHTML

Specifically developed for use in combination with the re-architectured and re-written iText 7 PDF platform, the pdfHTML add-on makes full use of the enhanced and new capabilities of iText 7. pdfHTML supports more HTML tags and CSS features such as floating and fixed positioning, and @media rules and queries. It’s also easier to extend for custom tags. It integrates seamlessly with other iText functionalities such as barcodes, PDF/A and PDF/UA output, and advanced typography features of pdfCalligraph. pdfHTML has more robust handling of imperfect or invalid HTML input.

XML Worker

XML Worker was designed and developed based on the previous version of iText – iText 5. As such, it is limited by the constraints of the iText 5 PDF engine – challenged by the growing needs of the users. XML Worker was fundamentally designed to be top-to-bottom, and text line based, which has its limitations when dealing with the block level concepts and the flexible layout options of HTML. It also lacks generic handling of CSS styles, support for media queries (e.g. screen vs print), and a convenient extension mechanism (custom HTML elements, custom CSS handling)

Integration in today’s document workflow

Today, it is in the interest of HTML/CSS experts to be able to inject their documents and files in the wider corporate document workflow, which is often PDF based. It should be quite clear this is possible without the HTML/CSS experts acquiring an in-depth or advanced knowledge of the ins and outs of PDF syntax and quirks. With the iText 7 PDF platform, assisted by the pdfHTML add-on, this insertion of HTML/CSS documents into the corporate document workflow can be a ‘fire and forget’ automated process. It allows for HTML/CSS integration in the world of applications that are document and workflow oriented, supporting mission critical company applications.

About iText

iText is the world’s foremost platform to create PDF files and integrate them in corporate applications. Originally developed by Bruno Lowagie, iText is available as an open source product with community support, as well as a commercial product, worldwide supported by the iText Group NV (with offices in Europe, North America and Asia).
In 2016, the popular iText 5 version was succeeded by the fully re-written and re-architectured iText 7 platform, with advanced add-on options. iText represents a unique capability of facilitating the use of dynamic and complex PDF documents in corporate document workflows, throughout the whole lifecycle of the documents.