A powerful template engine that converts data into iText quality PDFs

White Paper V1.2

Introduction

Many organizations strive to make templating more user-friendly, without constantly pestering the development team with ad hoc requests. What if you could have an easy way to configure and design data-driven PDF output templates or input forms, and connect these to the appropriate back-end logic and data, ensuring correct business results and professional output?

That's what iText DITO is all about.

What is iText DITO?

iText DITO simplifies the process of creating multiple output templates or input forms. An intuitive WYSIWYG (What You See Is What You Get) front-end gives you the ability to create your own data-driven output templates or input forms, tied to a powerful back-end. iText DITO handles everything from HTML5 input form creation and data processing with JSON, to the generation of PDFs – all without requiring developers to write any code. This provides a way to get results while reducing workload for developers and saving you time and resources.

iText DITO is an enterprise-grade template engine built on proven technology; using iText 7 Core with its pdfHTML and pdfCalligraph add-ons to enable seamless integration of HTML5, CSS 3, PDF 2.0 and global language/ligature support into your document workflows.

In this white paper, we discuss the how and why of iText DITO. Starting with a broad overview explaining the features of the framework, we then look at what iText DITO can do, and how it can help you. This document is aimed at developers who will have to implement and maintain this solution. Therefore, we'll go into extensive technical detail, explaining advanced features, configuration options and workflow implementation.

Why use iText DITO?

There is a clear need in the market for a solution that offers users the power of dynamic data-driven form creation, allowing them to quickly design PDF output templates or input forms.

iText DITO is a powerful template engine that converts data into iText quality PDFs. This solution allows users without technical expertise to design an output template or input form and link it to a desired output document, using a system set up by developers for data processing.

At its core, iText DITO bridges the gap between an input document and the desired output format, using the data that has been gathered. This means you're not limited to forms; the engine can accept any data that can be converted to JSON and output to either PDF, or structured data that can be written to other databases. All this enables you to have a truly data-driven document workflow leading to significant savings for your business.

How iText DITO Works: A high level overview

iText DITO is a template engine with two main components: a user-friendly browser-based application and a Java SDK. The Java SDK can be configured by developers to do the heavy lifting, while the browser-based application gives users an easy to use tool to leverage the power of the Java SDK. The application itself can be configured to fit the needs of the business, and the workflow can be integrated into an existing IT environment.

Using the browser-based application users can quickly create HTML5 input forms and output templates which can be utilized in a number of ways, such as input forms to capture and submit data entered by users (for example an application form). Alternatively, you might want to populate an output template using existing data from a database, to generate PDF invoices or other documents. You can also link input forms to output templates, to be altered or combined with other data before being output to a specific style and saved as a PDF.

The templates are built using easy to add elements such as rich text, text fields, sub-forms and tables. Styling can be added to those elements in the iText DITO browser-based application, however, if you require further styling then you can simply add your own CSS to templates. For example, company formatting and branding can be applied to the templates created in iText DITO.

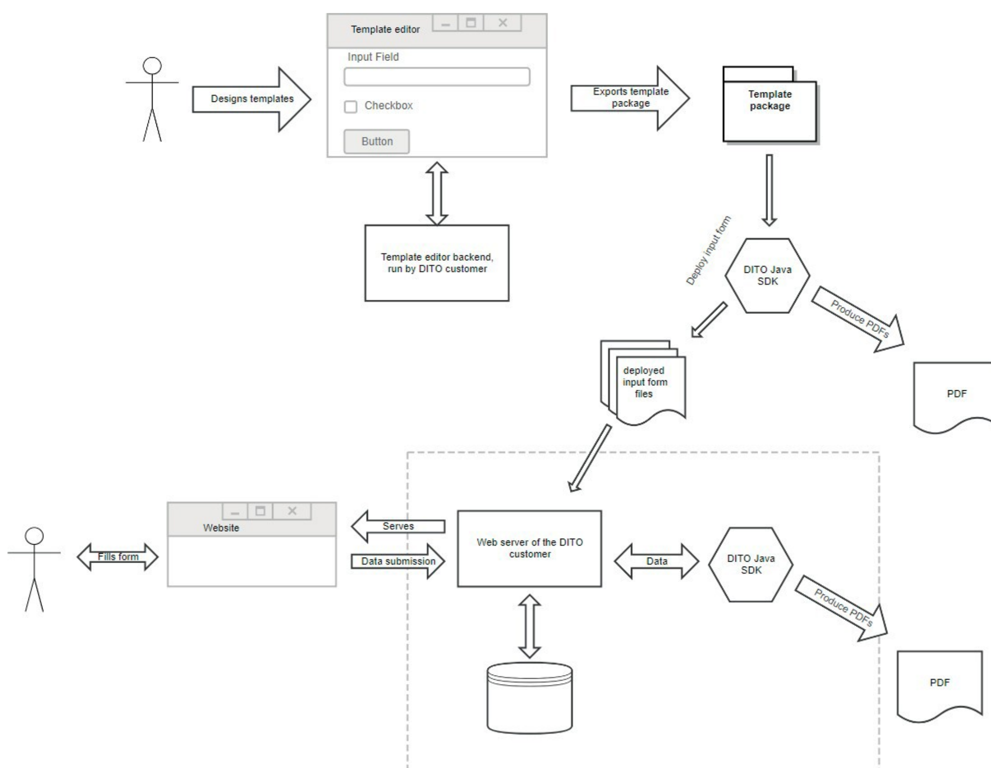
All input fields can be linked to a variable identifying the input field. This enables the user to combine fields into calculations (e.g. Total = Price x Quantity), but also allows output documents to be created easily. Some input data will be relevant as output, some might be relevant as a source for calculations, and some might even only be used for authentication and serve no further purpose.
Data binding makes it easy to include input from non-human sources such as databases or RESTful API calls. iText DITO uses data converted into JSON format turning it into much more than a form creation tool; it can also be used to structure data from different sources into output documents. You can of course also combine user input with additional data from other sources.

When the user is satisfied with both the input form and output template, they can export the result to an iText DITO template package. This is essentially an archive file containing all the necessary information to build the forms, including all relevant resources such as fonts and images. To use the input form or output template, you need the iText DITO SDK which is available either as a Docker container for use with non-Java environments, or as a native Java library if you prefer. The SDK is the engine powering iText DITO and is responsible for both publishing and executing the logic to move from input data to structured output data. The output usually takes the form of a PDF, but thanks to the data binding it's also possible to export the input, for example to store it in a database.

The SDK runs using the iText DITO template package and a set of other parameters. It can be wrapped into a standalone application (which might be run from the command line), but usually the functionality of the SDK will be integrated into existing software. This might translate to an easy-to-use button that users can click without the need for any assistance from developers, apart from the initial integration and implementation.

iText DITO enables a powerful end-to-end workflow where any user can easily create output templates or input forms using data from users, databases or both. They can then apply the necessary logic to get end results that suit their needs. This can be a PDF such as an invoice or a legal record, or even new entries in a database.

You only need to integrate iText DITO and upgrade the implementation as necessary (for example with new CSS styles). Both developers and users can work far more efficiently, requiring far less of each other's time.



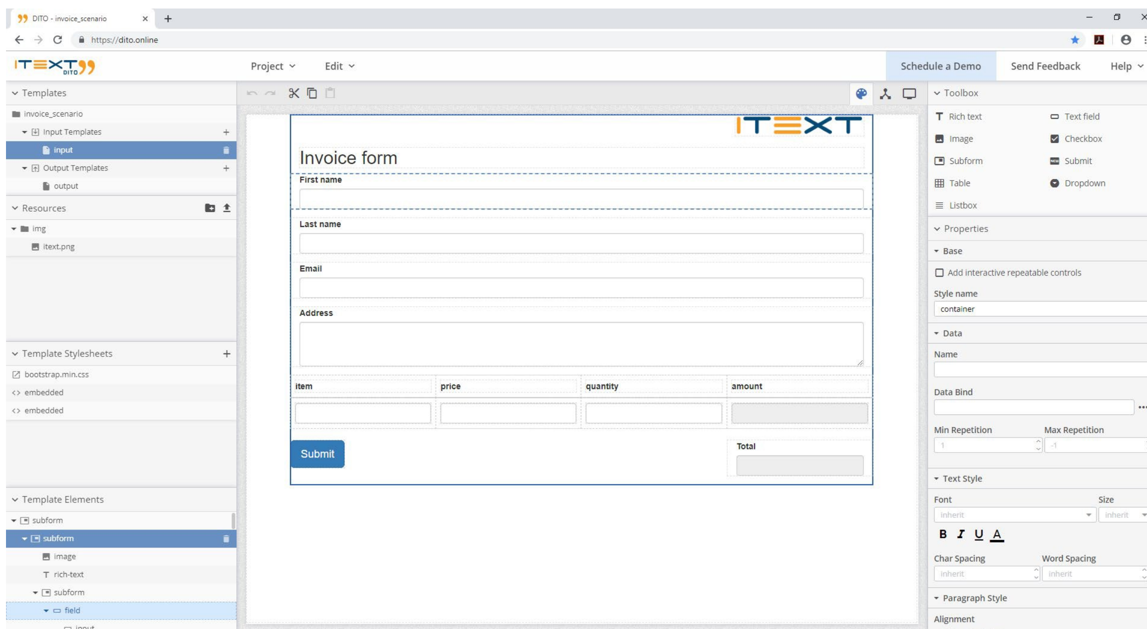
Advanced functionality deep dive

The editor

The power of iText DITO comes in part from the web application. The easy-to-use yet powerful WYSIWYG interface allows even non-technical users to create forms that operate exactly as needed. It is the logical connection between input and output, and the comprehensive way in which to create that logic, that really makes the product unique.

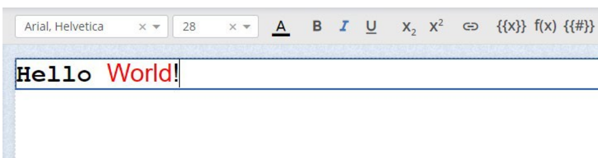
The editor is used to make proprietary iText DITO template packages. They contain all the necessary resources to produce the needed output, in PDF or written to an existing database.

The application enables anyone to create unique templates. It can run locally, but the web app is usually accessed through a local server. The editor consists of a central workspace; on the right are subform-tools such as table, image and data binding controls, and on the left is a column for additional finetuning of the templates, with template stylesheets and elements also displayed.



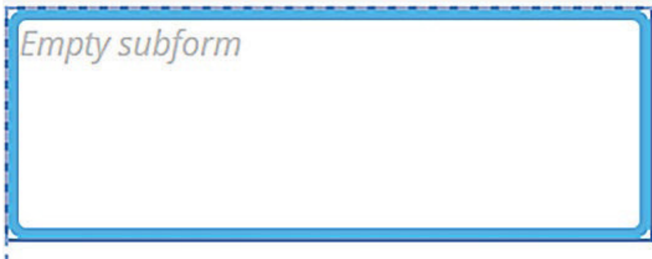
RTF

Any user familiar with text editor software will immediately feel at home in the central workspace area. Rich text options allow them to easily adjust the styling, font, color and outline of text. Text can be both static and dynamic. It's possible to insert dynamic intrusions taken from data to customize each output form. Obvious examples are invoice numbers, or names of recipients taken from an existing database.



Subforms

Central to the functionality of any form are the subforms. A subform is a group containing different elements. By editing the subform, all elements within it are affected. It is possible to directly affect the visual style of a form through subforms. When creating an invoice form, for example, the user can group the item name, number, quantity and price in a subform and create a border around all these elements. Visually, this makes it easier for anyone filling the form to keep an overview. Furthermore, the subforms allow for some powerful dynamic features, as we'll discuss further in this document.



Tables

iText DITO also supports tables. This feature is especially useful when combined with data binding of the table fields. It allows iText DITO to get input data from existing databases, or output data gathered through the forms to other databases. If there are multiple input values available for a table row, it can be repeated as often as necessary. A user filling out the form can manually add new rows, but iText DITO can also automatically add new items as needed when getting input from a JSON file.

In the example of the invoice form, you can imagine a table containing an item name, a price and a quantity in each row. If you want to use iText DITO to create PDF invoices from a database, you'll be confronted with different customers ordering different amounts of items. If a customer orders a laptop at a cost of 500 dollars, and a keyboard at a cost of 10 dollars, the table can automatically create a second row containing the second item.

Item	Price
Laptop	$500.00
Keyboard	$10.00

Text

A text field is a very straightforward addition to any form. It allows the user to type whatever they want. First and last names are straightforward examples. A user can create placeholder text that will appear in an unfilled text form box, to clarify what is expected. They can also attach styles or specify data bindings.

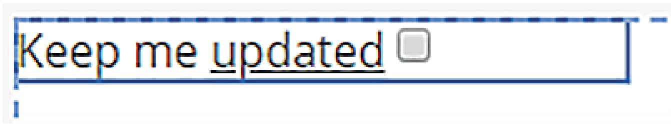
First Name

Images

As a form should usually reflect the company style, adding images is an essential feature. You can add images from the internet or upload your own. Uploaded images will be archived in the iText DITO template package after creation of the template. The most popular use case for adding images is the inclusion of a company logo.

Checkbox

The addition of checkboxes is rather straightforward. Of course, it's possible to specify the label and the style accompanying the check box.



Submit button

The submit button is only available for input templates and can't correspond with an element on the output template. As the name suggests, the button triggers the collection of the data from the form. When the button is clicked, the data is sent to a previously selected URI. The data can now be used to produce a PDF output template, but it can also be stored in a database. You can easily customize the appearance of buttons to fit the template style.

Input versus output

The workflow for creating the output form is almost identical to the creation of the input form. It's possible to omit certain input or perform calculations on input fields in order to generate output fields. By adjusting the style, the output form quickly becomes a formal document that stands on its own and doesn't just look like a cheap replica of the input form. How different the input and output forms look depends entirely on your business needs.

Formatting and logic

The brain of iText DITO is the SDK which uses many components of the iText 7 code, although it is available as a separate product. The software is built on the Open Web Platform stack, but more specifically, HTML, CSS and JavaScript are the foundations of iText DITO. This powerful SDK parses iText DITO templates created with the editor and has a lot of advanced functions which make it ideal for data processing. In this segment, we'll discuss in some detail the way iText DITO handles data and logic, and how you can add dynamic formatting. Some of the features below are quite advanced and are intended for implementation by technical users. We provide the following documentation for the benefit of developers.

Data binding

Data binding connects data with template elements. Binds make it possible to use data dynamically; to create output forms using data from input forms, to store data in databases, and to automatically input data from non-human sources.

Suppose you want to create custom tickets for an event, and you already have all the necessary information of the invitees stored in a database. You want to create an input form to gather the relevant data from the database, and an output form that looks like the final invitation you'd be sending. This output form is a PDF, built from the data in the database.

The first thing you need to add is a first name. You need to create JSON data defining the data bind property, followed by the field value. The JSON-data could look like this:

```
{
"first_name": "John"
}
```

`first_name` is the value of the data bind property of a field, while the field value taken from the data source is `John`. If you specify a field containing text input and bind that input to `first_name`, the field will automatically resolve to `John`.

Data bind properties

Properties

Data

Data Bind

time

Display Format

date('HH:mm:ss', 'UTC')

Bind Format

date('YYYY-MM-DD HH:mm:ss', 'Europe/Brussels')

When we bind values from data to a template it goes through the following transformation:

```
data value -> raw value -> display value
```

The `raw value` is the internal format that iText DITO uses.

The display format covers the second transformation while the bind format is responsible for the first one. The same syntax is used for both.

Bind format should be specified when data is expected in a format that is different from the internal format. In the example above, iText DITO reads the date-time from data stored in Brussels time zone format and displays the time in UTC.

Resolving logic

iText DITO resolves input following a set of rules. It is important to know these rules in order to get the results you want, when using JSON input data with a form template.

Resolving order

The data is automatically resolved against the data bound to the nearest element in a parent chain, or to the root data if no parents in the chain are bound to data. Suppose you've created a field with a binding `(first_name)`, nested within a subform without binding, nested in turn within a parent subform with a binding `(person)`.

```
subform ("person" binding)
|
--- subform (no binding)
|
--- field ("first_name" binding)
```

Now imagine the JSON data for input looks as follows:

```
{
"person": {
"first_name": "John"
},
"first_name": "Jane
```

In this example, the field will resolve to `John`, as the algorithm looks for the nearest bound parent, in this case the subform person. You can change this behavior by adding the `$` sign in front of a data bind value. The parameter tells the algorithm to look for the binding relative to the data root. So, the form structure would look like this:

```
subform ("person" binding)
|
--- subform (no binding)
|
--- field ("$.first_name" binding)
```

The field would resolve with the `first_name` binding closest to the root of the JSON data example, which is `Jane`.

Resolving multiple data entries

Resolving data to data bound fields is easy when there is exactly one entry that corresponds with one field. More often than not, the number of data entries is variable. You might have one customer who ordered two different items, and another who ordered ten. If you want to automatically create invoices from an existing data set containing the orders, your form needs to cope with this variable input.

Specifically, you would need the relevant subform to automatically multiply as many times as needed. That way, you only have to add a single subform with entry fields for item, quantity and price. If a customer orders more than one item, a new line is added, as many times as needed.

By adding the `[*]` parameter to the data bind value of a field in the web editor, the algorithm knows it needs to multiply a subform as many times as the input data requires. Apart from the parameter, the structure remains the same.

```
subform ("people[*]" binding)
|
--- field ("first_name" binding)
--- field ("last_name" binding)
```

Now the subform will multiply for each data entry. If the input data specifies three `first_name` and `last_name` binds like in the example below, the above subform will be added to the form three times.

```
{
"people": [
{ "first_name": "John", "last_name": "Doe" },
{ "first_name": "Jane", "last_name": "Diluca" },
{ "first_name": "Han", "last_name": "Pak" }
]
}
```

The [*] parameter means you never have to add the same subform multiple times manually, while also giving your template the flexibility to handle an unknown amount of data entries.

If you want to publish a form for manual filling, you can check a check box in the web editor to add + and - buttons next to a subform.

Calculations and logic

We've already talked about performing calculations in iText DITO. Sometimes, it might be impossible to show the necessary information on the output form using only data binding. You might for example want to show a total at the end of an invoice, containing the sum of all items ordered in their respective quantities, multiplied by the VAT.

In order to simplify processing of the data without the need for pre-processing, iText DITO can handle many straightforward calculations. Calculations allow you to assign values to certain fields depending on the values of other fields. It's possible to calculate a field using other fields which are the results of calculations themselves. In a sense, you can treat iText DITO like a simple spreadsheet.

In order to perform calculations on the variables in a field, you must name the fields. The field names act as variables that you can use in mathematical expressions. If you have a field named `price` and another one named `quantity`, you can create a field showing the total of an order by defining it as `price * quantity`.

The [*] parameter can be used in this context as well. The parameter allows you to perform a calculation using a variable number of fields. If, for example, you want to calculate a grand total based on the order of an unknown number of different items, you'd need the sum of the price field of all those items. You can do this by using the following notation:

```
sum(price[*])
```

iText DITO supports the following functions:

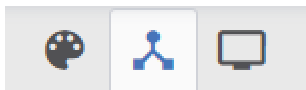
- `sum`
- `product`
- `concat`

And the following operators:

```
"+", "-", "*", "/"
```

Data Mapping Mode

The editor has an additional view mode to see and configure data binding. You can switch to the data mapping mode by clicking the following button in the editor:



This will give you a view like this:

Visual styles and formatting

We already talked about some easy-to-use visual options within the web editor, but iText DITO has far more up its sleeve. You can define very specific personal styles using CSS code, and you can even make a single form locally responsive based on the region in which you want to use it.

Value formatting

Value formatting functions are powerful tools that enable you to display values in specific ways in the output form. You can use the functions to specify the desired format. Currently, iText DITO supports number and currency formatting.

Number formatting is represented as a string, preceded by the num operator. A formatting string might look like this:

```
num('$#,##0.00', 'de-de')
```

In the above example, there are two arguments. The first one specifies the pattern; the second argument adds a locale. A locale-parameter is optional, but very handy when you want to deploy a single form in multiple geographical regions. It affects grouping, decimal separators and currency symbols.

For example, a format of `#,###.00` and a text of `4500.20` will look as follows with a locale of “de”:

```
4.500,20
```

And as follows with a locale of “us”:

```
4,500.20
```

There are six special characters you can use to define formatting. All other characters in a formatting string will be left intact.

- `0 (Digit)`
- `# (Digit, zero shows as absence)`
- `. (Decimal separator)`
- `- (Negative sign)`
- `, (Grouping separator)`
- `% (Percent, multiplies number by 100)`

Time/date formatting

For time/date formatting, as with number formatting there are two arguments that specify the pattern, and the (optional) time zone ID. If no time zone ID is specified, then the local one will be used.

An example showing the local time and date in Paris would be formatted as follows:

```
date('YYYY-MM-DD HH:mm:ss ZZ', 'Europe/Paris')
```

The `ZZ` in this example displays the UTC (Coordinated Universal Time) offset from the selected time zone.

The pattern consists of special characters as well as any other characters, special characters can be escaped with [] brackets.

Special characters:

Month:

- M - 1 2 ... 11 12
- Mo - 1st 2nd ... 11th 12th
- MM - 01 02 ... 11 12
- MMM - Jan Feb ... Nov Dec
- MMMM - January February ... November December

Quarter:

- Q - 1 2 3 4
- Qo - 1st 2nd 3rd 4th

Day of Month:

- D - 1 2 ... 30 31
- Do - 1st 2nd ... 30th 31st
- DD - 01 02 ... 30 31

Day of Year:

- DDD - 1 2 ... 364 365
- DDDo - 1st 2nd ... 364th 365th
- DDDD - 001 002 ... 364 365

Day of Week:

- d - 0 1 ... 5 6 do - 0th 1st ... 5th 6th
- dd - Su Mo ... Fr Sa ddd - Sun Mon ... Fri Sat
- dddd - Sunday Monday ... Friday Saturday

Day of Week (ISO):

- E - 1 2 ... 6 7

Week of Year:

- w - 1 2 ... 52 53
- wo - 1st 2nd ... 52nd 53rd
- ww - 01 02 ... 52 53

Week of Year (ISO):

- W - 1 2 ... 52 53
- Wo - 1st 2nd ... 52nd 53rd
- WW - 01 02 ... 52 53

Year:

- YY - 70 71 ... 29 30
- YYYY - 1970 1971 ... 2029 2030
- Y - 1970 1971 ... 9999 +10000 +10001

Week Year:

- gg - 70 71 ... 29 30
- gggg - 1970 1971 ... 2029 2030

Week Year (ISO):

- GG - 70 71 ... 29 30
- GGGG - 1970 1971 ... 2029 2030

AM/PM:

- A - AM PM
- a - am pm

Hour:

- H - 0 1 ... 22 23
- HH - 00 01 ... 22 23
- h - 1 2 ... 11 12
- hh - 01 02 ... 11 12
- k - 1 2 ... 23 24
- kk - 01 02 ... 23 24

Minute:

- m - 0 1 ... 58 59
- mm - 00 01 ... 58 59

Second:

- s - 0 1 ... 58 59
- ss - 00 01 ... 58 59

Fractional Second:

- S - 0 1 ... 8 9
- SS - 00 01 ... 98 99 SSS - 000 001 ... 998 999
- SSSS ... SSSSSSSSS - 0000.. 0010.. ... 9980.. 9990..

Time Zone:

- z/zz - EST CST ... MST PST
- Z - -07:00 -06:00 ... +06:00 +07:00
- ZZ - -0700 -0600 ... +0600 +0700

Unix Timestamp:

- X - 1360013296

Unix Millisecond Timestamp:

- x - 1360013296123

Notes on time/date formatting:

For month formatting, using the MMMM pattern will not result in four-letter months, but will instead return full month names.

If an incompatible time is parsed (one with two conflicting hours declarations for example) then an exception will be thrown

If the pattern doesn't contain more specific information (e.g. it contains a year but doesn't contain a month) then for more specific information the initial value will be used (first month, first day etc.)

If the pattern doesn't contain more broad information (e.g. it contains a month and day but doesn't contain a year) then the closest possible local information will be used (e.g. 2019 for the year etc.)

If the pattern has a UTC (Coordinated Universal Time) offset then it will be used, and the time zone will be ignored.

Due to DST (daylight saving time), there is a possibility that a time either does not exist, or it has previously existed.

If you try to parse a time that never existed, it will skip forward by the amount of the DST gap (usually 1 hour).

If you try to parse a duplicated time, then the earlier instance will be taken.

Stylesheets and classes

iText DITO templates make use of the Open Web Platform. The templates are therefore to a great extent compatible with CSS stylesheets. Advanced users can customize almost all visual properties of a template with CSS. However, since the iText DITO web editor is meant to be used by non-technical users, CSS knowledge is not required. Anyone can modify the visual styling with the designer tool, but CSS of course offers more flexibility.

You can also embed personalized stylesheets while integrating iText DITO into the company infrastructure. You can add the relevant styles once, allowing the user to select the style needing to understand CSS.

You can add a new embedded stylesheet or use an external one. The stylesheets can be linked with elements through element classes. It's also possible to specify styles without explicitly linking to them. For example:

```
*[data-dito-element="rich-text"] {
color: red;
```

In this example, all the text in rich text elements would default to red, unless the user explicitly overrides the text color by changing the lay-out.

Installation and integration

iText DITO is not an out-of-the-box application. The framework needs to be integrated with an existing IT infrastructure. While anyone can use the framework to create templates and forms, the integration needs to be done by an experienced developer. The developer can define optimal workflows by integrating with the existing IT infrastructure.

iText DITO is only available for Java and requires Java 8 to run the web editor application. Forms are published and data is processed with the Java SDK. The SDK also produces the PDF output. iText guarantees software compatibility with both the Oracle JRE and Open JRE. You need version 8 with update 152. iText DITO might not work properly on older versions.

Web editor installation

You can download the JAR file from the iText Artifactory. In order to run the web editor application, you'll need a configuration file such as this one:

```
server:
  applicationConnectors:
- type: http
  port: 8080
adminConnectors:
- type: http
  port: 8081
logging:
  level: INFO
workDirectory:
  path: D:/DITOWork
```

As you can see, the editor uses two ports. The first one (8080 in the above example), is used to access the web editor itself. Users of the application will connect through this port.

The second port (8081) is used for access to administrative features. You can of course configure the ports manually, as long as the editor and the administrator connector use a different one. You should make sure the admin port isn't accessible from outside the local trusted network.

The `workDirectory` parameter specifies the directory that will be used for temporary storage. This parameter is required for the web editor application to work correctly, make sure the path is valid.

iText DITO uses the Dropwizard webserver in the back-end. If you need documentation detailing how to set up the https connection and how to add https certificates, you can refer to the Dropwizard configuration manual, available online.

In order to run the editor application, you need the following command:

```
java -jar dito-server.jar {server, check} config.yml
```

The server command starts the web application. As soon as it's started, it should be available through the specified port. The check command makes sure the configuration file offers valid configuration data. If something is not working properly, you'll see the errors printed.

As an administrator, you can access the health check feature. Health checks help you identify the cause of problems with the application. The feature is available through the administrator port:

```
http://localhost:8081/healthcheck
```

The result of a health check is returned in JSON format.


```
{"deadlocks":{"healthy":true},"workDirectory":{"healthy":true}}
```


Assuming everything works well, you'll see `{“healthy”:true}` for all checks. If there is a problem, the JSON data will contain a message describing the problem. For example:


```
"workDirectory":{"healthy":false,"message":"Work directory does not exist"}
```


Java SDK

The Java SDK is the heart and brains of the iText DITO framework. The SDK deploys the input form, produces output PDFs from output templates, processes calculations and formatting, and processes input and output from and to external databases.

Installing with Maven

The Java SDK is available as a Maven module. In order to install it, you need the necessary dependency from the iText Artifactory. First, you need to add the repository to the Maven POM file as follows:

```
<repositories>
<repository>
<id>dito</id>
<name>DITO Repository</name>
<url>https://repo.itextsupport.com/dito</url>
</repository>
</repositories>
```

Subsequent deployment is easy. Just add the dependency to your POM file like this:

```
<dependencies>
<dependency>
<groupId>com.itextpdf.dito</groupId>
<artifactId>sdk-java</artifactId>
<version>${dito.sdk.version}</version>
</dependency>
</dependencies>
```

The `dito.sdk.version` statement is a variable specifying your current version of the framework. The value can for example be `1.0.0`.

Installing with Gradle

To reference the Java SDK from Gradle, you need the necessary dependency from the iText Artifactory. First, you need to add the repository to the Gradle build file as follows:

```
repositories {
        mavenCentral()
        maven {
                url "https://repo.itextsupport.com/dito"
        }
        maven {
                url "https://repo.itextsupport.com/releases"
      }
}
```

Referencing the dependency is now easy. Just add the corresponding entry to the `dependencies` section:

```
dependencies {
    compile group: 'com.itextpdf.dito', name: 'sdk-java', version: ditoSdkVersion
}
```

The `ditoSdkVersion` statement is a variable specifying your current version of the framework. The value can for example be `1.0.0`.

Workflow integration

Every iText DITO workflow starts with the creation of a form template, exported to an iText DITO template package. This file will be at the center of the rest of the workflow. You'll probably want to deploy input forms, accept data submissions for those forms, and finally create output PDFs using the input data, following the output template.

Loading the license key

Before you can use the Java SDK, you need to install the license key. This is easily managed by using the following commands:

```
File licenseKeyFile = "...";
DitoLicense.loadLicenseFile(licenseKeyFile);
```

For volume license customers

We advise querying your remaining volume frequently and sending a warning message to your own application/system before the limit is reached and stops working.

For customers using the native iText DITO Java SDK you can call `getRemainingProducePagesEvents` using the API and this function will return the number of pages remaining on your volume license key. If you have a perpetual license, this call will return `null`.

For iText DITO SDK for Docker users it is even simpler. Simply call `/api/license` to display the `expirationDate` of your license key as well as the remaining volume.

Input form deployment

As soon as a user has created a template package using the editor, the package can be deployed in your web server environment. The Java SDK will process the package into an input form.

You need the `InputTemplateDeployer` class from the `com.itextpdf.dito.sdk.input` package. Four parameters are relevant. You need to specify the path to the template package you want to use, the name of the input template inside the package, an alias for the input form and an output folder. The deployed form will be stored in this location.

```
File projectFile = new File("templatePackage.dito");
String templateName = "input";
File deployFolder = new File("deployed");


InputTemplateDeployer.deployTemplateFromPackage(projectFile, templateName, deployFolder);
```

Inside the folder specified for deployment, you can find an input file. This file contains the actual form you've created. The file name will be identical to the template name, as specified in the `TemplateName` argument. As the form is now created, you're free to rename it into anything you like.

The folder will also contain a subfolder named `js.` Inside, the Java SDK has put the files necessary to support interactive features of the form. If applicable for the template you've used, you'll also find a `resources` folder here. You can host all files directly from the deployment folder.

iText DITO allows for flexible publication of the form. You can, for instance, create a dedicated web page and direct users there, or you can embed it inside an existing web page using an `<iframe>`.

Accepting data from forms

When designing the template, you can specify a URI. Data collected by the form, will be sent there using POST or PUT, as specified during the design of the template.

The type of the data payload is `application/x-dito-data-submission`, but you can easily convert the data to a type that's easier to work with. For now, only JSON is supported. Data submissions are handled by the `DataSubmission` class from the `com.itextpdf.dito.sdk.submission` package. The example below illustrates how to get submission data in JSON format from the `InputStream`, which represents the payload:

```
String json = DataSubmission.fromInputStream(is).asJsonString();
```

PDF creation

One of the primary functions of iText DITO is the automatic creation of stylish and useful PDFs based on data gathered by the input form. If the template package contains an output template, all you need is the `PdfProducer` class from the `com.itextpdf.dito.sdk.outputpackage`. Creating a PDF using an output template and JSON data might look like this:

```
File templatePackageFile = new File(templatePackagePath);
String templateName = "output";
FileOutputStream fos = new FileOutputStream(new File(outFilePath))
String json = "{...}";

PdfProducer.convertTemplateFromPackage(templatePackageFile, templateName, fos, new
JsonData(json));
```

You can also combine accepting data submissions and creating PDFs. That way, a PDF will be created for each filled out form. The example below shows how you can automate the workflow:

```
String json = DataSubmission.fromInputStream(is).asJsonString();
File templatePackageFile = new File(templatePackagePath);  String
templateName = "output";
FileOutputStream fos = new FileOutputStream(new File(outFilePath))
String json = "{...}";

PdfProducer.convertTemplateFromPackage(templatePackageFile, templateName, fos, new
JsonData(json));
```

Using iText DITO

We have now covered the iText DITO framework quite extensively. We've also discussed the need for a data-driven solution like iText DITO, the possibilities of the framework, and we've touched on the most important technical aspects. Below, you'll discover a simple example of a workflow using iText DITO, using as an example the creation of an invoice template.

A use case example

An invoice template is a straightforward example of an iText DITO use case. You can easily imagine a sales representative in the field using a tablet to input orders for a customer. The sales rep would need an easy tool at their fingertips, while the customer requires a professional looking and easily understandable invoice for payment.

In this scenario, the head of sales can use iText DITO to develop a form for use by the sales rep. The dynamic form allows them to add products to an order, specify how many products are ordered, and expand the order with multiple products as necessary. By completing the input form, the sales rep can then place the order.

Of course, it wouldn't do to send the customer a screenshot of the form. Even though it looks clean and easy to use, it's nowhere good enough to serve as an invoice. Furthermore, a real invoice needs to have specific information included, such as an account number or addresses. Lastly, the customer expects their invoice to arrive via email in PDF format, as PDF offers a widely compatible yet unchangeable digital file that carries with it a sense of authority.

With iText DITO, the order made by the sales representative is automatically converted to a PDF file that's ready to be emailed to the customer. What's more, iText DITO handles the data gathered through the input form in such a way that it can be shared with other parts of the company's IT infrastructure. Imagine the order going straight to the order pickers in the warehouse, for fast delivery.

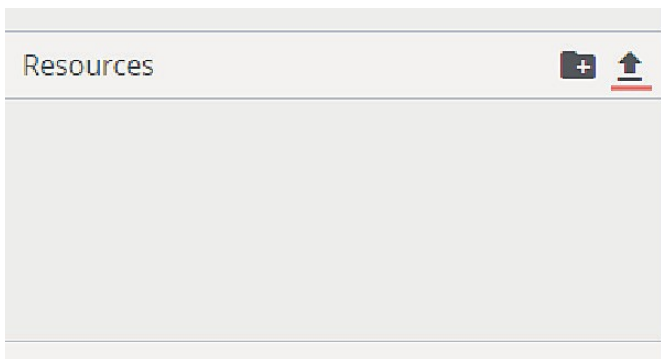
When iText DITO has been implemented correctly, the head of sales would not need the help of a developer to create both the form and the PDF output. They could easily build a template for both, following the company style and formatting.

Comprehensive walkthrough

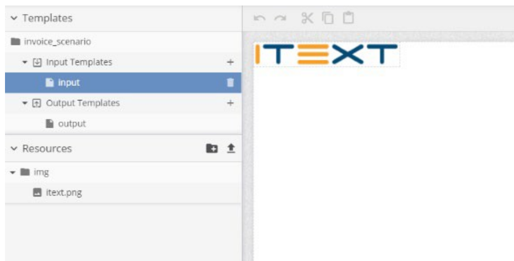
Now, let's have a look at how the above use case would translate into a practical workflow. The template we are creating might be used by a sales person. They can fill out the form with orders for their customer. After submitting the order, iText DITO will automatically create a professional looking invoice in PDF. We'll use sample data to try out our form:


```
{
"invoice_no": "12345", "company_name":
"ABC, Inc.", "address": "12 West street,
NY, USA", "items": [
{ "name": "Blue pen", "price": "20.1", "quantity": 10 },
{ "name": "Black pencil", "price": "0.1", "quantity": 1 },
{ "name": "Red pen", "price": "20", "quantity": 78 },
{ "name": "Blue pencil", "price": "0.15", "quantity": 13 }
]
}
```

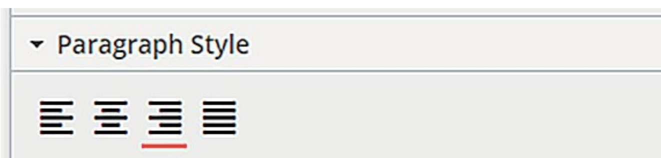

The creation of the template starts in the web editor. We start our new project by uploading the company logo. On the left of the editor, you can find the resources panel. Here, you can either add an image from the web or upload your own.



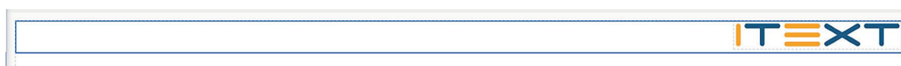
As soon as the image is added, you can drag it to the editing area. The editor will automatically create a new element containing the image.



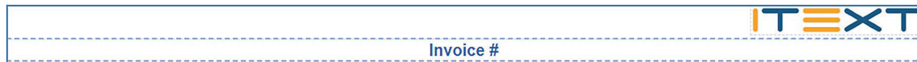
Next, we align the image to the right of our form. In order to do so, we must first wrap it into a subform. Click the subform icon in the toolbox and drag the image into the subform. Now we can apply the right text alignment from the paragraph style properties.



The result looks as follows:

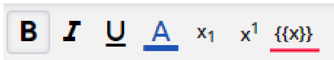


The invoice we are creating will have a unique number, shown at the top of the form. In order to add this header, we need a new static text element. You can easily add it and use the style controls to edit the visual appearance of the text.



Our form must take the invoice number from the input data. In order to do so, we insert a data binding intrusion with the value `invoice_no`.

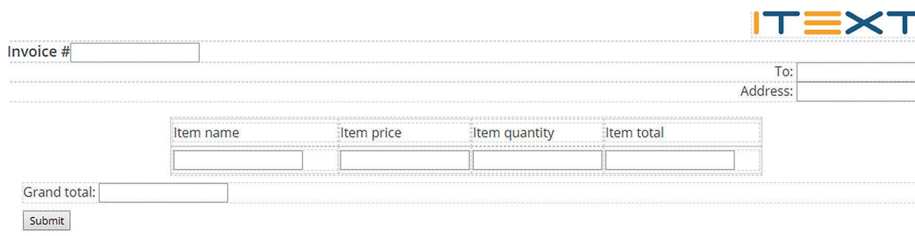
Click the underlined button on the right of the toolbar to add the binding.



You can add static text elements containing the company name and address in a similar way. We align the contact information to the right, so it sits neatly underneath the logo.

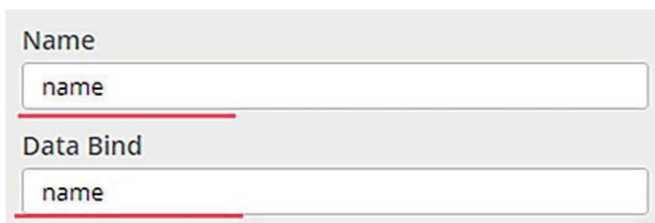
Since we already have some data binding going on, it's a good idea to do a preview. At the top of the editor window, you can find the preview button. iText DITO needs sample data for the preview So we can use the data provided at the beginning of this segment. The JSON data contains `"invoice_no": "12345"` as the first entry. The value `12345` corresponds with the `invoice_no` variable, which we used as a data bind in our form. The preview shows the expected result:

In editing mode, it's time to add the bulk of our input form. We need a table containing the name of the item being bought, the unit price, the quantity of the order, and the total for that item. Underneath, we add a field with the grand total for the entire order.



The table above requires some configuration. We need to define the relationship between the individual fields and the input data, but also between the fields themselves. We not only need to apply data binding to the fields, but we must also name them. That way, we can invoke the value of the fields as variables for further calculations.

The item name, item price and item quantity fields all need a data bind and a name. You can use `name, price` and `quantity` for both. The example below shows how the properties of the item name field need to be filled.



The item total field will be filled using data provided by the `item price` and `item quantity fields`. Therefore, it does not need a data bind. The field does need a name though, as we are going to perform calculations on in the `Grand total` field. We call it `item_total`. Under `Calculation`, we select `product`. In order to calculate the total amount owed, the field must contain the product of `price` and `quantity`.

Name

item_total

Data Bind

Calculation

product × price, quantity

Before we can move on to the `Grand total` field, we need to tell our form that each item from the input data needs to have its own row in the table. In other words: the table needs to extend itself with additional rows as many times as there are item data entries in the JSON data. This makes the form flexible; it will be able to handle orders containing only one item as well as orders containing multiple items.

Select the second table row (the one containing the fields) from the elements tree on the left, and specify the multiplied `items[*]` binding.

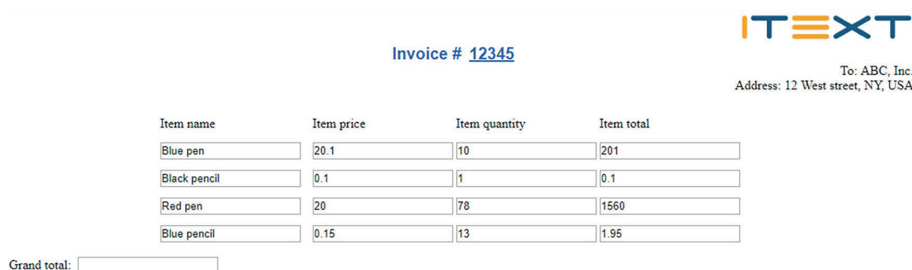
▾ Data

Name

Data Bind

items[*]

Another preview will illustrate the result of this binding. As there are four entries under Items, the form will automatically add four rows in the table. The Item total will also be filled as a product of quantity and price for each item.



ITEXT

Invoice # 12345

To: ABC, Inc.
Address: 12 West street, NY, USA

Item name	Item price	Item quantity	Item total
Blue pen	20.1	10	201
Black pencil	0.1	1	0.1
Red pen	20	78	1560
Blue pencil	0.15	13	1.95

Grand total:

This works perfectly, assuming all input comes from previously gathered data. Of course, our sales representative should be able to use the form manually to put in a client's order. If we want them to be able to add multiple items to one invoice, we need to add interactive controls. These controls will appear next to a row in the table. By clicking '+', the representative will be able to add a new row manually. With the '-' button, they can remove a row.

Select the second row of the table again and look for the Add interactive repeatable controls option. Click the checkbox to enable the feature.

▾ Base

☑ Add interactive repeatable controls

Lastly, we need a submission button. By clicking this button, data will be sent to the destination of our choice. Add the button, and try previewing the form without any input data, so you can see the vanilla form that will be presented to the sales rep.

ITEXT

Invoice #

To:
Address:

Item name	Item price	Item quantity	Item total
	€0,00		€0,00

Grand total: €0,00

Submit

The form still looks rather crude, but all the required functionality is there. Now you can try and polish the form, or you can apply custom CSS styles to really wrap it in the company style.

Now you can copy the input template to the output template. You must get rid of the buttons to manually add or remove rows, and the submit button is also useless here. The output form will be converted to PDF and serve as an official invoice, so it should look accordingly. You can preview the output with the sample data to see how everything will look.

When you're satisfied, you can export the template to an iText DITO template package ready for use by the Java SDK.

Conclusion: Taking action with iText DITO

iText DITO is a powerful tool that empowers business users while decreasing the time developers must spend performing nonessential tasks. In businesses today, users want to use technology to achieve a goal. They know what they need but may lack the technical knowledge and background to translate their vision into reality. Developers have this knowledge but aren't always familiar with the specific needs of the end user. Both of them trying to work out a solution to a problem isn't very efficient.

The main goal of iText DITO is to provide users with the tools to turn their own data-driven forms vision into reality, without needing to involve a developer. Valuable time gets saved, while the pace of innovation is increased.

There still is an important role for the developer in the early stages of the implementation of iText DITO. They must do the actual legwork of integrating the framework in the existing IT infrastructure and configuring it to be used. But after this initial configuration, the tool is ready to go.

iText DITO is the only framework available today that empowers end users to create advanced and functional forms combined with PDF output. With iText DITO installed, users can create a variety of forms in the look and feel of their choice. The input of the forms is easily structured, can be output to a database, and of course converted to PDF to be either used or archived. Sound interesting?

Contact us to learn more at: https://itextpdf.com/en/products/itext-dito.

About iText

iText is the world's foremost platform to create PDF files and integrate them in corporate applications. Originally released in 2000, 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 document lifecycle.