



A powerful **low-code document generation engine** that  
converts data into iText quality PDFs

White Paper V1.3

## 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, 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. An intuitive WYSIWYG (What You See Is What You Get) front-end gives you the ability to create your own data-driven output templates, tied to a powerful back-end. iText DITO handles everything from HTML5 template 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 better results faster, while empowering your business users and reducing the workload for developers.

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 not only written for template designers, but also for professionals with a more technical profile who will implement iText DITO. 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 document generation, allowing them to quickly design PDF output templates.

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, using a system set up by developers for data processing.

## How iText DITO works: A high level overview

iText DITO is a document generation product with two main components: a browser-based template editor and an SDK. The SDK is available as either a Docker Container image or as a native Java application. The SDK can be configured by developers to do the heavy lifting and fit the needs of the business, and integrate the workflow into an existing IT environment.

Meanwhile the browser-based editor gives users an easy to use tool; leveraging the power of the SDK to quickly create templates which can be populated with existing data from a database and generate PDF documents.

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 existing data sources such as databases, RESTful API calls or web forms that collect user input.

When the template designer is satisfied with the output template, he or she can export the result to an iText DITO template package. This is essentially an archive file containing all the necessary information to build PDF documents, including all relevant resources such as fonts and images. To use the 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 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 using data from users, databases or both. They can then apply the necessary logic to get end results that suit their needs.

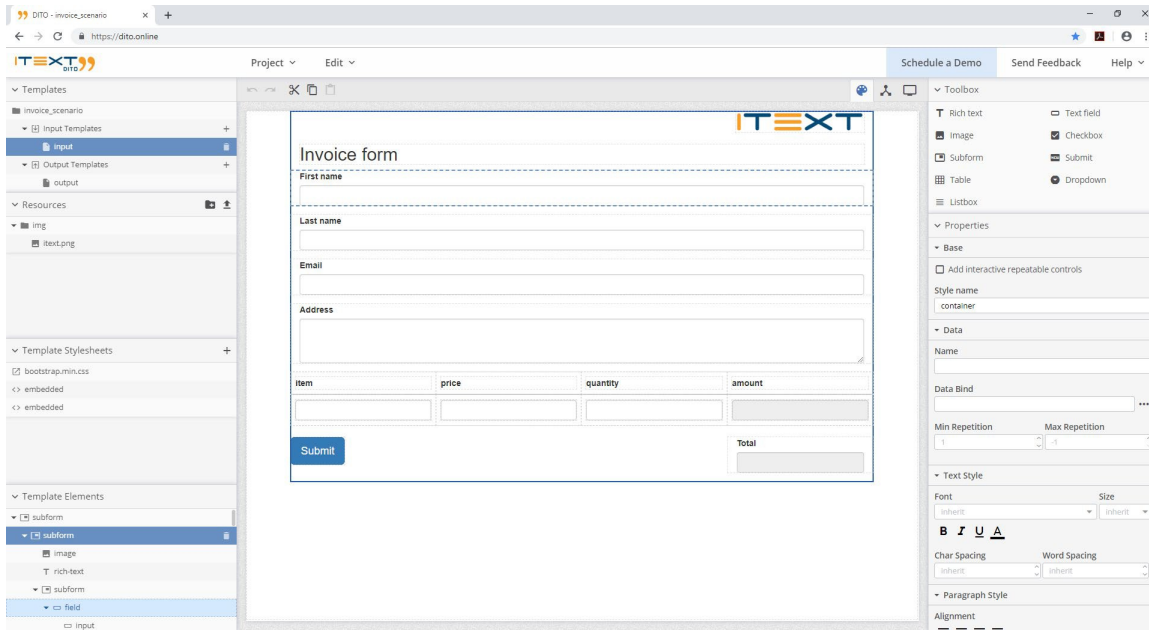
You only need to integrate iText DITO and upgrade the implementation as necessary (for example with new CSS styles). Both developers and users can work far more efficiently, requiring far less of each other's time.

# Advanced functionality deep dive

## The iText DITO Editor

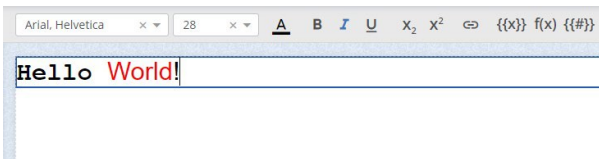
The power of iText DITO comes in part from the web application. The easy-to-use yet powerful WYSIWYG interface allows even non-technical users to create templates that operate exactly as needed. The editor is used to make proprietary iText DITO template packages. They contain all the necessary resources to produce the needed output in PDF.

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.



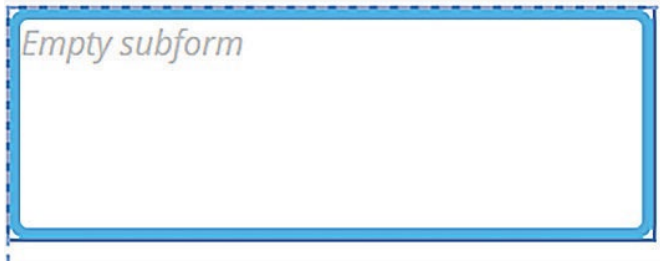
## RTF

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. You could imagine that for instance a certain section of the envisioned output document will have different margins. By including all elements (rich text, tables, images, etc.) of that section into a single subform, you can manage the margin settings at the subform level.



## Tables

iText DITO also supports tables. This feature is especially useful when combined with data binding of the table fields. If there are multiple iterations of the bound field, iText DITO supports repeating a table row as many times as there are iterations of the bound field.

In the example of an invoice document, 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

## Images

As a document should usually reflect company branding, 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. A typical use case for adding images is the inclusion of a company logo.



## Formatting and logic

iText DITO's powerful SDK parses templates created with the editor and has a lot of advanced functions which make it ideal for data processing. In this section, we'll discuss in some detail the way iText DITO handles data and logic, and how you can add dynamic formatting and content insertion.

### Data binding

Data binding connects data with template elements. Binds make it possible to use data dynamically; the data binds serve as placeholders for data that is merged with the template at document generation time.

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 personalized tickets in PDF file format, that contain the data in the database.

If you would like to ticket to show the registered person's first name, the JSON data to be merged with the template should contain a key-value pair that holds this information. The JSON data could for instance look like this:

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

In the template, you can insert a rich text element that contains a data bind to the first name field in the JSON file, which will serve as a placeholder for that specific data element. At generation time, the value that corresponds to the bound key will be taken from the JSON file referenced in the document generation instruction.

### Data bind properties

When you bind a value from data to a template it goes through the following transformations:

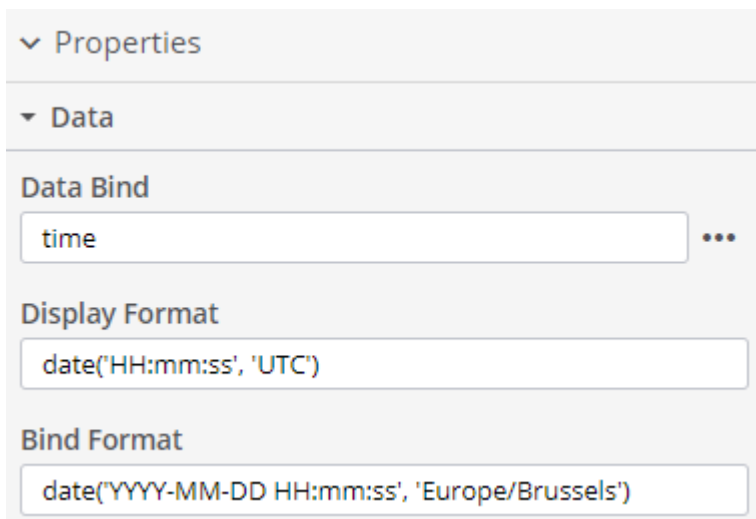
data value -> raw value -> display value

The **raw value** is the internal format that iText DITO uses for the data type in question. The display format covers the second transformation while the bind format is responsible for the first one.

Even though the format of your data may be identical to iText DITO's raw format (e.g. for date values the internal format is ISO 8601), it is still recommended to explicitly specify the bind format.

#### Example 1

In this first example below, iText DITO reads the date-time from data stored in Brussels time zone format and displays the time in UTC.



The screenshot shows a 'Properties' panel with a 'Data' section. Under 'Data Bind', the value 'time' is entered in a text box. Below it, the 'Display Format' is set to 'date('HH:mm:ss', 'UTC')'. The 'Bind Format' is set to 'date('YYYY-MM-DD HH:mm:ss', 'Europe/Brussels')'. Each text box has a three-dot menu icon to its right.

#### Example 2

In this second example iText DITO will convert a Zulu date-time notation, which is frequently used as a timestamp format, into a custom date-time format.

<b>Data Bind</b>	invoice_date	...
<b>Display Format</b>	date('dddd, MMMM Do, YYYY')	
<b>Bind Format</b>	date('YYYY-MM-DDTHH:mm:ss.SSSSSSZ')	

The JSON value '2020-05-14T00:00:00.000000Z' will be displayed as 'Thursday, May 14th, 2020' in the resulting document.

## 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 (loops)

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 or table row to automatically multiply (loop) as many times as needed. That way, you only have to add a single subform or table row with 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 or table row 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 or table row multiple times manually, while also giving your template the flexibility to handle an unknown number of data entries.

### Filtering loops

You can create tables with repeating rows for any subset of a data array. Imagine that you have a list of line items in an invoice data set, and in the envisioned PDF document you want to build different tables, for instance to distinguish between line items of different types.

The filter can be set via syntax. The syntax to define the filtered loop looks like this:

**`iteratingElement[[conditionExpression]]`**

in which `iteratingElement` is the root element of the array and `conditionExpression` defines the filter.

To set the filter, you can use the operators in the table below. String values can be contained within single or double quotes.

Operator Name	Type	Syntax	Expression Example
Is equal to	Text/Number	==	<code>{{fieldname}}=="fieldvalue"</code>
Is not equal to	Text/Number	!=	<code>{{fieldname}}!="fieldvalue"</code>
Is empty (string length is zero)	Text	<code>isEmpty()</code>	<code>isEmpty({{fieldname}})</code>



Is blank (no characters or only blank characters)	Text	isBlank()	isBlank({{fieldname}})
Is not empty	Text	!isEmpty()	!isEmpty({{fieldname}})
Is not blank	Text	!isBlank()	!isBlank({{fieldname}})
Contains	Text	contains()	contains({{fieldname}}, "fieldvalue")
Does not contain	Text	!contains()	!contains({{fieldname}}, "fieldvalue")
Is greater than	Number	>	{{fieldname}}>fieldvalue
Is greater than or equal to	Number	>=	{{fieldname}}>=fieldvalue
Is less than	Number	<	{{fieldname}}<fieldvalue
Is less than or equal to	Number	<=	{{fieldname}}<=fieldvalue
And	Logical	&&	contains({{fieldname}}, "fieldvalue1")&&!contains({{fieldname}}, "fieldvalue2")
Or	Logical		{{fieldname}}=="fieldvalue1"  {{fieldname}}=="fieldvalue2"

The operators in the table can be combined with the calculation functions that iText DITO supports, such as `sum()`, `product()`, `concat()` and the newly added `size()`. The `size()` function calculates the count of items in an array. This can be combined with the filtering option for arrays to count how many items a filtered subset of an array contains.

### Example

The example below illustrates how you can use the filtered loop feature to build tables that only contain a specific subset of items from an array. The data set in this example contains an array of items:

```
"items": [
  {"item": "Commercial Invisibility Cloak License .NET", "type": "Software License", "price": "6800", "quantity": "1", "lineitemtotal": "6800"},
  {"item": "Updates and Support Commercial Invisibility Cloak .NET", "type": "Support Plan", "price": "666", "quantity": "1", "lineitemtotal": "666"},
  {"item": "Commercial Non-Production House Elf", "type": "Software License", "price": "3140", "quantity": "1", "lineitemtotal": "3140"},
  {"item": "Updates and Support Commercial Non-Production House Elf", "type": "Support Plan", "price": "152", "quantity": "1", "lineitemtotal": "152"},
  {"item": "Nimbus 2001", "type": "Controller", "price": "5640", "quantity": "13", "lineitemtotal": "73320"},
  {"item": "Fantastic Beasts and Where to Find them", "type": "Book", "price": "28", "quantity": "1", "lineitemtotal": "28"}
]
```

To build a table that only contains items of the type "Software License", you can bind a repeating table row to the following expression:  
`items[{{type}}=="Software License"]`

Software Licenses			
ITEM DESCRIPTION	PRICE	QUANTITY	TOTAL
{{item}}	\$:{{price}}	{{quantity}}	\$ {{product({{price}}* {{quantity}})}}

Name

Data Bind

Min Repetition

Max Repetition

When previewed with the above data sample, the table resolves to only two iterations of the repeating row, one for each of the items of the type Software License.

# Software Licenses

ITEM DESCRIPTION	PRICE	QUANTITY	TOTAL
Commercial Invisibility Cloak License .NET	\$ 6800	1	\$ 6,800
Commercial Non-Production House Elf	\$ 3140	1	\$ 3,140

## 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:

Function	Example Values	Expected Result
<code>sum()</code>	1, 3, 5	9
<code>product()</code>	1, 3, 5	15
<code>concat()</code>	1, 3, 5	135
<code>size()</code>	1, 3, 5	3 (counts the number of items in the array)

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:

Schedule a Demo Send Feedback Help

Your statement

contact tel +44 20323328259  
see reverse for call times  
Fax: +44 2032333219  
used by deaf or speech impaired customers  
www.moneybank.co.uk

as Drive Arlington, TX 76011

Account Summary	
Opening balance	{{opening_balance}}
Payments in	{{payments_in}}
Payments out	{{payments_out}}
Closing balance	{{closing_balance}}
Account type	{{account_type}}

Sort Code	Account Number	Sheet Number
{{sort_code}}	{{account_number}}	{{sheet_number}}

Debit Value	Credit Value
{{debit}}	{{credit}}
{{sum(sum({{\$.items[*].credit}})-sum({{\$.items[*].debit}}))}}	

Properties

Data

Display Format: num('\$.##0.00', 'de-de')

Calculation: sum sum({{\$.items[\*].credit}})-sum({{\$.items[\*].debit}})

Data Samples +

- data.json
- account\_name: James J. Clark
- account\_number: 30127972
- sort\_code: 40-05-15
- account\_text: 502 3//2 4562 00684 00506 2000...
- opening\_balance: 602200000
- payments\_in: 90000000
- payments\_out: 510201000
- closing\_balance: 502941139
- sheet\_number: 00007
- account\_type: CURRENT/EUROPEAN CURRENCY
- [ ] Items
- { } 0
- transaction: 19-11-2012
- value\_date: 04-06-2006
- transaction\_details: ABC Stores Unlimited
- debit: 108.2

## Conditional visibility

iText DITO allows you to define logic in your output templates to insert content conditionally in generated PDF documents. Conditions can be either based on text or on numbers.

### Text-based Conditions

Text-based conditions evaluate whether a textual data element (e.g. the country of residence of a customer) has a specific value (e.g. United States) or not. If at runtime the condition resolves as true, then the selected element or substring will be included in the document. If the condition resolves to false, the conditional selection will be omitted, and the rest of the document will reflow.

The example below shows the three-step process of creating a text-based condition, using the condition settings wizard in the iText DITO Editor.

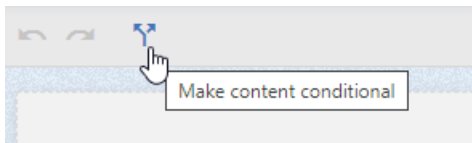
#### Step 1: Select any element or substring

The first step in the process consists of indicating which piece of content should be inserted conditionally. This can be an element from the template elements tree or a substring of a rich text element. Switch to data mapping mode via the command bar and make your selection on the editing canvas.



#### Step 2: Make content conditional

With the element or the substring selected in the data mapping mode, click the double arrows icon in the command bar at the top of the screen. This will activate the condition settings wizard that will guide you through the process of defining the text-based condition.



### Step 3: Define the condition

In the wizard that opens up, you can select the data field that you want to evaluate, use it as text, select the operator and insert a value to compare it against. In the example below you will check whether the value of the field `{{customer.country}}` is equal to "United States". If that condition returns true, then the rich text element selected in step 1 will appear in the generated document, if the condition does not return true, it is left out.

A screenshot of a "Condition Settings" dialog box. The dialog has a title bar with a close button (X) in the top right corner. Below the title, it says "Show the content in the selected area if the following condition is true". There is a "Wizard" toggle switch which is currently turned off, and a "Syntax" label with a refresh icon. The "Field" section has a text input containing "customer.country" and a three-dot menu icon. The "Use as" section has a dropdown menu with "Text" selected. The "Operator" section has a dropdown menu with "Is equal to" selected. The "Value" section has a text input containing "United States". At the bottom, there are two buttons: "Delete condition" (grey) and "Save" (blue).

In the resulting document, when generated for a customer that does not reside in the United States, the selected fragment will not be included.

# INVOICE

**Malfoy, Inc.**  
421, Castle Road  
East Bedfordshire, 84751  
United Kingdom

Invoice Number: 14852478

**Total Due**  
\$ 89,320

Invoice Date  
12 May 2020

Due Date  
11 June 2020

ITEM DESCRIPTION	PRICE	QUANTITY	TOTAL
Commercial Invisibility Cloak License .NET	\$ 6,800	1	\$ 6,800
Updates and Support Commercial Invisibility Cloak .NET	\$ 666	1	\$ 666
Commercial Non-Production House Elf	\$ 3,140	1	\$ 3,140
Updates and Support Commercial Non-Production House Elf	\$ 152	1	\$ 152
Nimbus 2001	\$ 5,640	13	\$ 73,320
Fantastic Beasts and Where to Find them	\$ 28	1	\$ 28

*This is an electronic invoice.  
Please think twice before you print it.*

**Thank you for your business**

### Payment info

Account No.: 139 2458 7842 4526 851

<b>SUBTOTAL</b>	\$ 84,106
<b>DISCOUNT (10%)</b>	\$ 8,411
<b>TAX AMOUNT (18%)</b>	\$ 13,625
<b>GRAND TOTAL</b>	<b>\$ 89,320</b>

This text will appear conditionally, for UK-based customers

## Number-based Conditions

As well as text-based conditions, you can also evaluate a field against a numerical value. Depending on the validation outcome, selected content will be inserted in the generated documents or left out.

By means of example, in an invoice template you can make a table row that contains a discount amount appear only if a discount is given. The process has three steps.

### Step 1: Select any element or substring

In the example below the selection is a table row element.

- T rich-text
- ▼ table-cell
  - T rich-text
  - ▼ table-cell
    - T rich-text
    - ▼ table-row 🗑
      - ▼ table-cell
        - T rich-text
        - ▼ table-cell
          - T rich-text

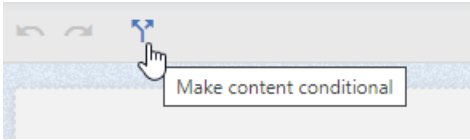
ITEM DESCRIPTION	PRICE	QUANTITY	TOTAL
{{item}}	{{price}}	{{quantity}}	\$ {{product({{price}}*{{quantity}})}}
<b>SUBTOTAL</b>			\$ {{undiscounted_total}}
<b>DISCOUNT (10%)</b>			\$ {{discount}}
<b>TAX AMOUNT (18%)</b>			\$ {{tax_amount}}
<b>GRAND TOTAL</b>			<b>\$ {{grand_total}}</b>

*This is an electronic invoice.  
Please think twice before you print it.*

**Thank you for your business**

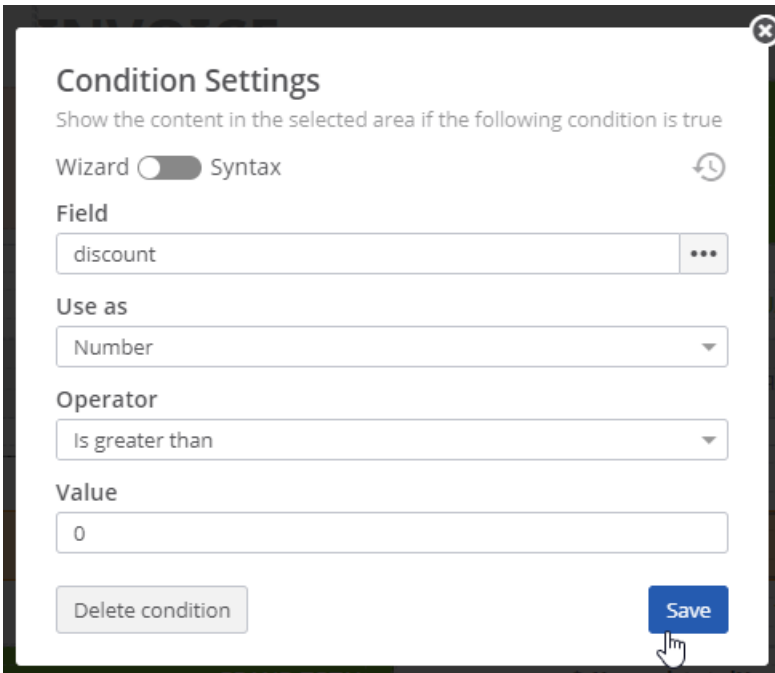
### Step 2: Make content conditional

With the table row element selected in data mapping mode, click the double arrows icon in the command bar at the top of the screen. This will activate the condition settings wizard that will guide you through the process of defining the number-based condition.



### Step 3: Define the condition

In the wizard that opens up, you can select the data field that you want to evaluate, use it as `Number`, select the operator and insert a value to compare it against. In the example below you will check whether the value of the field `{{discount}}` is greater than 0. If that condition returns true, then the table row element selected in step 1 will appear in the generated document, if the condition does not return true, it is left out.



If a document is generated with this template based on a data set that has a value which is not greater than 0 for the `{{discount}}` field, then the entire row that contains the data binding of the discount will be omitted and the rest of the document will reflow accordingly.

### Conditions Pane

You can easily navigate the conditional sections in your template via the Conditions pane in the Data Connections panel that you see to the right of the editing canvas in data mapping mode. From this list of conditions you can edit or delete conditions at any time.



## 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 ... SSSSSSSS - 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.

To run the iText DITO Editor web application you require Java 8. Forms are published and data is processed with the iText DITO SDK which also produces the PDF output. To run the Java SDK you need version 8 with update 152. iText DITO might not work properly on older versions.

iText guarantees software compatibility with both the Oracle JRE and Open JRE.

### 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"}
```

## iText DITO SDK

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

The iText DITO SDK is available as either a Docker Container image or as a native Java version. For most applications and environments, we recommended using the Docker image for convenience. You can find instructions on downloading and configuration of the iText DITO for Docker SDK at <https://hub.docker.com/r/itext/dito-sdk>.

If you prefer to use the native Java version however, this section will detail how to download, install and configure the Java SDK using Maven or Gradle.

### 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 Artifacts. 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 Artifacts. 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 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 create output PDFs using the input data, following the output template.

### Loading the license key

Before you can use the SDK, you need to install the license key. iText DITO SDK for Docker users should refer to <https://hub.docker.com/r/itext/dito-sdk>, but for Java SDK users this is easily managed by using the following commands:

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

### Setting up proxy for volume license AWS calls (available since 1.0.10)

The proxy parameters for AWS calls can be configured with the environment variables `HTTPS_PROXY` (`https_proxy`) `HTTP_PROXY` (`http_proxy`) e.g:

```
HTTPS_PROXY=https://127.0.0.1:8888;HTTP_PROXY=http://127.0.0.1:8888
```

### 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 Java iText DITO 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.

## 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));
```

## 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 documents 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.

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-architected 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.