Specifying and populating documents in YAML with `lua-placeholders` in LaTeX

Erik Nijenhuis

Abstract

This article examines the implementation of the invoice template in GinVoice [3] and explores how the invoice template can better align with the LaTeX ecosystem by introducing an additional data layer in YAML using `lua-placeholders`. With the introduction of `lua-placeholders`, LaTeX users have complete freedom in formatting invoice templates, and the invoice templates are directly integratable with the enhanced version of GinVoice.

Keywords

LuaLaTeX, YAML

1 Introduction

During my work as a software engineer, I encountered a challenge for a company that drafts agreements and terms for multiple clients. One of the challenging aspects was keeping client data and regulatory documentation separate. Previously, I addressed this challenge in GinVoice [3] by generating additional LaTeX files with Python, which were then compiled alongside the main LaTeX file. However, this time, my goal was to provide a solution from within the LaTeX domain itself, rather than the application domain. The solution I developed, now known as `lua-placeholders` [5], introduces a shared data layer with YAML between LaTeX and application code. The package provides an intermediary layer specifically for data through YAML files. To demonstrate this solution, we use GinVoice as an example. This example, a Python GTK application that generates invoices with LaTeX, offers slightly more complexity and challenges than the legal domain has to offer.

1.1 The compiler — LuaLaTeX

I decided to use LuaLaTeX as the compiler for several reasons. Since 2016, I have been using LuaLaTeX, which greatly helped me with documents within computer science at the time. Over the years, I have gained a lot of experience in compiling with LuaLaTeX and see it as a suitable compiler as a developer, thanks to the ability to script in Lua, which I naturally appreciate as a programmer.

The ability to script in Lua offers several advantages. It allows me to perform complex tasks during the compilation process, such as processing YAML files or manipulating and structuring data. Additionally, LuaLaTeX supports Lua init scripts, allowing me to implement a custom compilation process with its own command line interface (CLI), further simplifying and optimizing the integration process for end solutions.

1.2 What is YAML?

As a DevOps engineer, I have often encountered YAML while working with tools such as Docker Compose, Travis CI, GitHub Actions, and Canonical's NetPlan (Ubuntu systems). YAML is widely used in the DevOps world for automating and managing configurations, functioning as a structured markup language for defining configuration files and capturing infrastructural and operational aspects of software applications.

YAML has become a crucial component of modern software development and deployment due to its simple syntax and flexibility. In combination with LaTeX, YAML provides a powerful mechanism for defining and managing structured data, which is particularly useful when integrating client data into LaTeX documents. Listing 1 shows an example of YAML used in conjunction with LaTeX.

```
supplier: grapefruit
client: juicing-joker
title: Grapefruit Inc. Invoice
subtitle: for fruits and stuff
currency: \$
number: 1
date: \today
...
```

Listing 1: `invoice-001.yaml`

2 GinVoice

In this section, we will take a closer look at GinVoice, an open-source Python GTK application that utilizes LaTeX behind the scenes to create invoices. Additionally, we will examine the provided invoice template and delve into the associated data within the invoice.

2.1 The application

GinVoice has multiple views. The most common is the main view, where you can draft multiple invoices simultaneously. In this view, depicted in figure 1, almost all components are visible. You can see the header, information tables, invoice rules, and the closing text included in it. Figure 1 shows that the input fields are already filled in, and their content does not deviate much from the end result, as seen in figure 2. Other application views will be discussed later in this section.

doi.org/10.47397/tb/45-1/tb139nijenhuis-placeholders

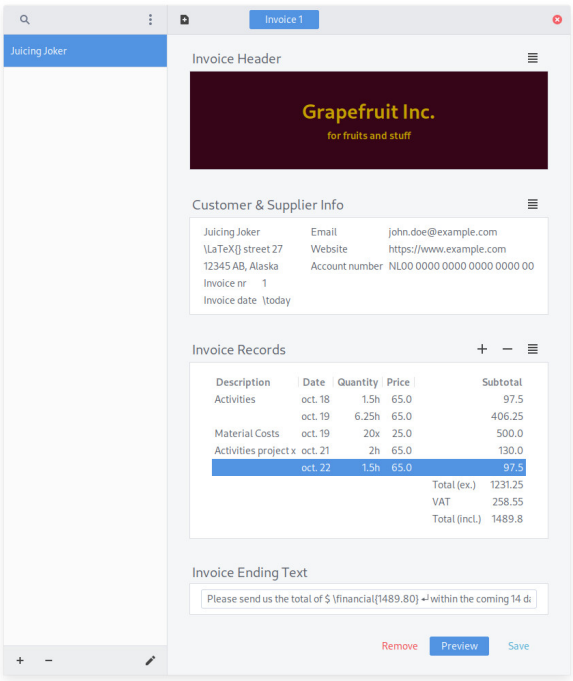


Figure 1: GinVoice — the application

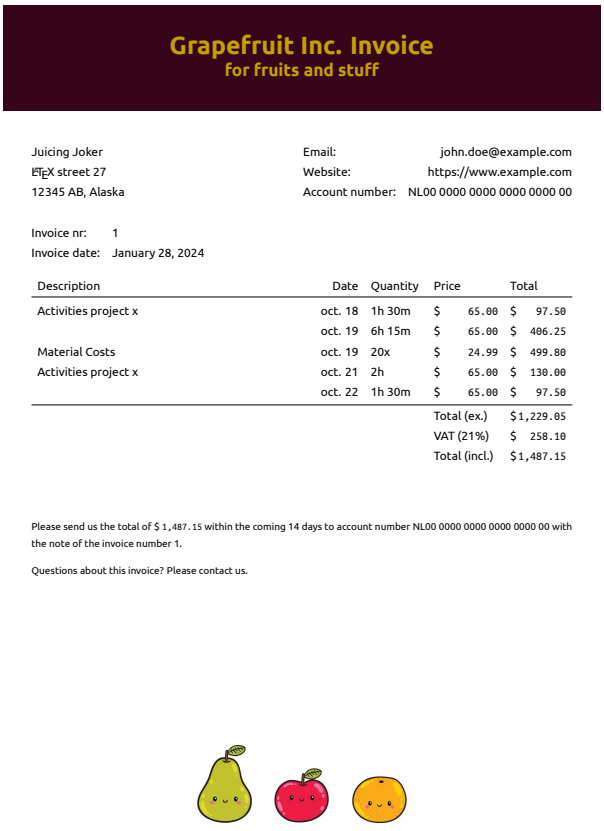


Figure 2: Sample invoice generated with GinVoice

2.2 LaTeX template

Below is an example of the code within the `document` environment:

```
\begin{document}
\thispagestyle{headermain}
\makeheader
\vspace{2cm}
\begin{tabular}{@{}l@{}}
    \begin{tabular}{@{}l@{}}
        \addressee
    \end{tabular} \\
    \begin{tabular}{@{}l l@{}}
        \customerinfo
    \end{tabular}
\end{tabular}
\hfill
\begin{tabular}{@{}l r@{}}
    \supplierinfo
\end{tabular}\\

\input{table}
\begin{invoice}{\columndef}{\tableheader}
                {\tablefooter}
    \tablerecords
\end{invoice}

{\footnotesize \theending{}}
\vfill
\begin{center}
    \images
\end{center}

\end{document}
```

Listing 2: `invoice.tex`

The source code in listing 2 demonstrates various macros that will be replaced by `lua-placeholders`: `\addressee`, `\customerinfo`, `\supplierinfo`, `\tablefooter`, `\tablerecords`, `\theending`, and `\images`. Additionally, there are variables such as title- and style-related information and `\currency` that will be handled.

2.3 Generated LaTeX files

It is important to note that GinVoice [3] currently uses a Python script, `generator.py`, to generate additional TeX files. These TeX files are then included in the template using `\include`, making the necessary macros available.

Starting with the language setting:

```
\usepackage[english]{babel}
```

Listing 3: `languages.tex`

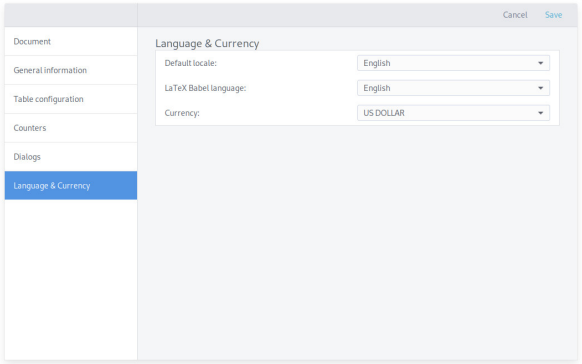


Figure 3: Language settings

At the time, I chose to include a separate language setting in the application, as shown in figure 3, so that words within the invoice are correctly hyphenated using `babel`.

Another aspect within the preamble is setting the document properties. These macros are imported from the generated file `meta.tex`, whose macros are later used in the `\hypersetup`.

```
\global\def\currency{\$}
\global\def\author{Erik Nijenhuis}
\global\def\title{Grapefruit Inc. Invoice}
\global\def\subject{Invoice for Juicing Joker}
\global\def\keywords{Invoice Grapefruit ↩
  Juicing Joker}
\global\def\producer{GinVoice Generator}
\global\def\creator{gingen}
\global\def\continuationheader{\title{} -- ↩
  \subject{}}
\global\def\continuationfooter{See next page.}
```

Listing 4: `meta.tex`

Common macros, such as `\title`, are used in multiple places. That is also why the `\title` does not need to be in the `header.tex`.

```
\global\def\subtitle{for fruits and stuff}
```

Listing 5: `header.tex`

The customer's address is placed in a macro, with the address lines separated by a newline.

```
\newcommand{\addressee}{Juicing Joker\\ ↩
  \LaTeX{} street 27\\12345 AB, Alaska}
```

Listing 6: `addressee.tex`

This approach would be suitable for a table with a single column or for, say, an `enumerate` environment.

The customer and supplier information assumes a table environment with two columns.

```
\newcommand{\customerinfo}{
    & \\
    Invoice nr: & 1 \\
    Invoice date: & \today \\
}
```

Listing 7: `customer_info.tex`

```
\newcommand{\supplierinfo}{
    Email: & john.doe@example.com \\
    Website: & https://www.example.com \\
    Account number: & NL00 0000 0000 0000 ↩
      0000 00 \\
    & \\
    & \\
    & \\
}
```

Listing 8: `supplier_info.tex`

The drawback of this setup is that an ampersand (&) does not have any function within the context of the macro itself. That would only be the case when working within a `tabular` environment. Despite most LaTeX editors giving an error for this, strangely enough, this approach still works.

The most significant challenge within the application was making the invoice table configurable. For this, there is a separate view, as seen in figure 4. In the figure, you can see that each column can have a different width, including length of text, maximum available space, or hidden. This added complexity from the application resulted in quite complex output in the generated `table.tex` file, as shown in the following code:

```
\newlength{\rowsize}
\setlength{\rowsize}{\linewidth}
\newlength{\cIsize}
\settowidth{\cIsize}{oct. 22}
\addtolength{\rowsize}{-\cIsize}
\addtolength{\rowsize}{-2\tabcolsep}
```

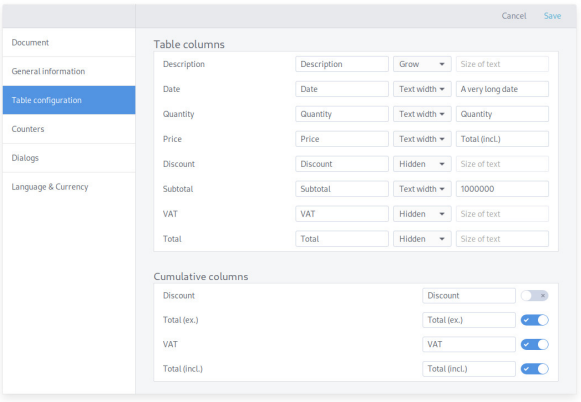


Figure 4: Table settings

```
\newlength{\cIIsize}
\settowidth{\cIIsize}{\textbf{Quantity}}
\addtolength{\rowsize}{-\cIIsize}
\addtolength{\rowsize}{-2\tabcolsep}
\newlength{\cIIIsize}
\settowidth{\cIIIsize}{\textbf{Total (incl.)}}
\addtolength{\rowsize}{-\cIIIsize}
\addtolength{\rowsize}{-2\tabcolsep}
\newlength{\cIVsize}
\settowidth{\cIVsize}{\$ 1,000.00}
\addtolength{\rowsize}{-\cIVsize}
\addtolength{\rowsize}{-2\tabcolsep}
\newcommand{\columncount}{5}
\newcolumntype\columndef ↩
  {L{1.00\rowsize-2\tabcolsep} R{\cIsize} ↩
  L{\cIIsize} F{\cIIIsize} F{\cIVsize}}
\newcommand{\tableheader}{\rowheadercolor ↩
  Description&\rowheadercolor ↩
  Date&\rowheadercolor ↩
  Quantity&\rowheadercolor ↩
  Price&\rowheadercolor Total\\}
\newcommand{\tablerecords}{
    Activities project x & oct. 18 & 1h 30m ↩
      & \currency\hfill\financial{65.00} & ↩
      \currency\hfill\financial{97.50}\\
     & oct. 19 & 6h 15m & ↩
       \currency\hfill\financial{65.00} & ↩
       \currency\hfill\financial{406.25}\\
    Material Costs & oct. 19 & 20x & ↩
      \currency\hfill\financial{24.99} & ↩
      \currency\hfill\financial{499.80}\\
    Activities project x & oct. 21 & 2h & ↩
      \currency\hfill\financial{65.00} & ↩
      \currency\hfill\financial{130.00}\\
     & oct. 22 & 1h 30m & ↩
       \currency\hfill\financial{65.00} & ↩
       \currency\hfill\financial{97.50}\\}
\newcommand{\cumoffset}{& & & }
\newcommand{\tablefooter}{\cum{Total ↩
  (ex.)}{1229.05}
\cum{VAT (21\%)}{258.10}
\cum{Total (incl.)}{1487.15}
}
```

Listing 9: `table.tex`

In addition to the complex column configuration, there are `\tablerecords` and `\tablefooter`, both similar to, for example, the supplier information.

The last generated file `footer.tex` defines the remaining missing macros, `\theending` and `\images`:

```
\newcommand{\theending}{Please send us the ↩
  total of \$ \financial{1487.15}
within the coming 14 days
to account number NL00 0000 0000 0000 0000 00
with the note of the invoice number 1.

Questions about this invoice?
Please contact us.}
\graphicspath{{/home/erik/share/ginvoice/img/}}
\newcommand{\images}{
    \includegraphics[width=.1\textwidth]{image1}
    \hspace{1.5em}
    \includegraphics[width=.1\textwidth]{image2}
    \hspace{1.5em}
    \includegraphics[width=.1\textwidth]{image3}
}
```

Listing 10: `footer.tex`

At the time, I chose to store all graphic files somewhere within the GinVoice environment. I linked this to LaTeX by using `\graphicspath`.

2.4 Invoice data

When looking at all the information coming from GinVoice, a few exceptions aside, we end up with the data presented in figure 5. For convenience, I have already divided all the information into separate entities, which will correspond to the YAML files, extensively discussed in the next section.

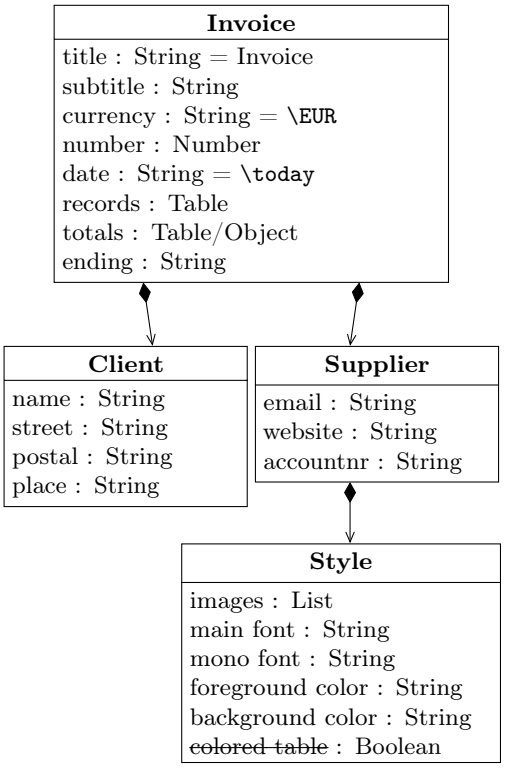


Figure 5: Class diagram of the invoice

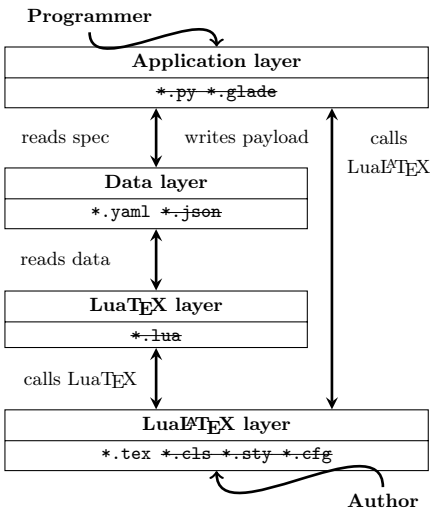


Figure 6: Levels within GinVoice

3 Invoice templates with `lua-placeholders`

This section demonstrates how YAML interfaces, also known as recipes, can be used as interfaces for invoice templates and how they can be linked to LaTeX.

The ultimate goal is to provide an efficient and customizable invoicing interface that can be easily integrated into an enhanced version of GinVoice. Figure 6 illustrates a representation of the new situation, with techniques irrelevant for this article crossed out.

Thus, the data, as seen in figure 5, is moved from the application level to the data level. This allows both Python programmers and LaTeX users to interact with the data level, something that is impossible in the current situation.

3.1 YAML specifications

Based on the data analysis in section 2.4, we can start working with the recipes. All recipes are placed in the `recipes` directory relative to the LaTeX project. Alternatively, you could store the `recipes` directory under `$TEXMFHOME/tex/` to make the recipes available everywhere.

3.1.1 The invoice

The invoice recipe, `recipes/invoice.yaml`, specifies two relationships: `supplier` and `client`, as mentioned earlier in section 2.4.

```
1 # Actors
2 supplier:
3   type: string
4 client:
5   type: string
```

Listing 11: `recipes/invoice.yaml`

How the corresponding recipes are loaded based on these values is described in section 3.2.3.

The data within the invoice part can optionally be standardized using a `default` field, as done for `title`. You can even invoke LaTeX from a default value, including other parameters using `\param`.

```
6 # Invoice variables
7 title:
8   type: string
9   default: Invoice \param{number}
10 subtitle:
11   type: string
12   placeholder: Subtitle
13 currency:
14   type: string
15   default: \EUR
16 number:
17   type: string
18   placeholder: Invoice number
19 date:
20   type: string
21   placeholder: Invoice date
```

In addition to default values, temporary placeholders can also be specified.

The most complex part of the invoice is the invoice table, where you can specify columns just like you do for other data types.

```
22 records:
23   type: table
24   columns:
25     description:
26       type: string
27     date:
28       type: string
29     quantity:
30       type: string
31       default: 0x
32     price:
33       type: number
34       default: 0
35     total:
36       type: number
37       default: 0
```

For most LaTeX users, the `total` column can be omitted and calculated using a package like `invoice2` [2]. To do that, it is also necessary to make the `quantity` field of type `number` and add an extra field like `quantity type`, so that you can display the correct notation for the `quantity` column.

For the final totals, I chose the type `object` so that I can manually set the different totals in LaTeX.

```
totals:
  type: object
  fields:
    total ex:
      type: number
      default: 0
    vat:
      type: number
      default: 0
    total incl:
      type: number
      default: 0
```

The final totals could also be handled in a more generic way, like the `extra fields` field in the supplier recipe (see section 3.1.3).

The last field of the invoice, `message`, uses the special YAML feature of multiline strings in the default value.

```
message:
  type: string
  default: |
    Please send us the total of ↩
      \currency~\paramfield{totals}{total ↩
      incl}
    within the coming 14 days to account ↩
      number
    \param[supplier]{account number} with ↩
      the note of the invoice number ↩
      \param{number}.\\[2em]

    Questions about this invoice? Please ↩
      contact us.
```

Using the pipe (`|`) activates this mode. This construction is ideal for large texts, possibly with LaTeX syntax.

3.1.2 Client

The client data does not have any special specifications compared to the invoice.

```
name:
  type: string
  placeholder: Client namme
street:
  type: string
  placeholder: Street + nr
postal:
  type: string
  placeholder: 9999 ZZ
place:
  type: string
  placeholder: City
```

Listing 12: `recipes/client.yaml`

Alternatively, all address details could be specified as a `list` type, along with a specification, as seen in `extra fields` in the supplier recipe. This would make the interface more generic but less adaptable within the LaTeX context.

3.1.3 Supplier

In the recipe for the supplier, the `style` field serves the same function as `supplier` and `client` of the invoice, allowing the user to choose which style to apply.

```
name:
  type: string
  placeholder: Supplier name
email:
  type: string
  placeholder: Email
website:
  type: string
account number:
  type: string
  placeholder: Account number
extra fields:
  type: table
  columns:
    key:
      type: string
    val:
      type: string
# Suppliers style
style:
  type: string
```

Listing 13: `recipes/supplier.yaml`

Another interesting field in this specification is `extra fields`. This field uses the `table` type to allow arbitrary additional information fields, such as the supplier's account number, VAT number, or any other relevant details. Using a table instead of a fixed number of fields gives the end-user the flexibility to add as much extra information as needed, without imposing restrictions.

3.1.4 Style

In the style recipe, fonts, colors, and multiple images can be specified. As mentioned earlier: for LaTeX users, this could be fully specified in LaTeX itself. The style recipe could then be omitted.

```
images:
  type: list
  item type: string
main font:
  type: string
```

```
  default: Ubuntu
mono font:
  type: string
  default: Ubuntu Mono
foreground color:
  type: string
  default: 000000
background color:
  type: string
  default: FFFFFF
```

Listing 14: `recipes/style.yaml`

A notable point here is the type for `images`, namely `list`. In section 3.3, you can see how this list is loaded at the bottom of the invoice.

3.2 The new invoice

Now that the recipes are in order, we can proceed to integrate them into LaTeX (in `invoice.tex`).

3.2.1 Loading recipes in the preamble

The recipes are loaded using the `\loadrecipe` macro.

```
\loadrecipe[\jobname]{recipes/invoice.yaml}
\loadrecipe{recipes/supplier.yaml}
\loadrecipe{recipes/client.yaml}
\loadrecipe{recipes/style.yaml}
```

For the `invoice` recipe, you can see that it is given a ⟨*namespace*⟩ of `\jobname` (the optional argument). This is because the `\param` macro by default uses `\jobname` as the ⟨*namespace*⟩, simplifying its use.

The other recipes do not specify a ⟨*namespace*⟩, meaning they use the 'basename' of the path as the ⟨*namespace*⟩. In this case, respectively, `supplier`, `client`, and `style`.

3.2.2 Currency

Regarding the currency, I have chosen to disguise it in the `\currency` macro. This is because it is also used in other files, such as `invoice.cls`.

```
\def\currency{\rawparam{\jobname}{currency}}
```

If the ⟨*currency*⟩ is not set, the default value from `style.yaml` is used. In this case, it defaults to `\EUR`.

3.2.3 Loading values

I've chosen to manage all YAML files related to the data in corresponding directories.

⟨*project name*⟩
- `recipes`
  - ⟨*recipe*⟩`.yaml`
- `invoices`
  - ⟨*invoice-xxx*⟩`.yaml`
- `clients`
- *etc.*

Values, also called the payload, are loaded similarly to recipes but with the `\loadpayload` macro. Due to the relationships described in section 2.4, it is slightly more complex than recipes because `lua-placeholders` does not offer anything standard for this.

```
\IfFileExists{invoices/\jobname.yaml}{
    \loadpayload[\jobname] ↩
      {invoices/\jobname.yaml}
    \strictparams
}{}
```

When loading invoice values, it is checked whether a corresponding YAML file exists. If so, that payload is loaded, and the experimental macro `\strictparams` is used, which means that errors will occur in the future if mandatory data is missing. If no corresponding file is found, an invoice template is compiled.

After loading the invoice data, we can check if a client is specified in the invoice data. We do this using `\hasparam`. This concerns the invoice data, for which we do not need to specify a ⟨*namespace*⟩.

```
\hasparam{client}{%
    \loadpayload[client] ↩
      {clients/\rawparam{\jobname} ↩
      {client}.yaml}
}{}
```

Generally, `\param` is not intended for use within the preamble because it can also yield placeholders with LaTeX markup. For such difficult situations, the macro `\rawparam` is written, as done for the client and supplier. This macro has no optional arguments; they often cause problems with, for example, `pgfkeys`.

```
\hasparam{supplier}{%
    \loadpayload[supplier] ↩
      {suppliers/\rawparam{\jobname} ↩
      {supplier}.yaml}
}{}
```

As you can see, loading the supplier does not differ from loading the client. However, there is a follow-up action after loading the supplier, namely checking if the style can be loaded. This is done in the same way as with the client and supplier themselves, but here you see that the ⟨*namespace*⟩ must be set.

```
\hasparam[supplier]{style}{%
    \loadpayload[style] ↩
      {styles/\rawparam{supplier}{style}.yaml}
    \setmainfont{\rawparam{style}{main ↩
      font}}
    \setmonofont{\rawparam{style}{mono ↩
      font}}
```

```
68     \definecolor{backgroundcolor}{HTML} ↩
         {\rawparam{style}{background color}}
69     \colorlet{bgcolor}{backgroundcolor}
70     \definecolor{foregroundcolor}{HTML} ↩
         {\rawparam{style}{foreground color}}
71     \colorlet{textcolor}{foregroundcolor}
72 }{}
```

For the style-related data, I chose to configure the values directly in the corresponding macros, such as `\setmainfont` and `\definecolor`, as long as a style is specified. You could also choose to set the style values by default based on the default values specified in the `style` recipe, by placing the configuration outside the `\hasparam` block.

3.3 Processing in the document

Before we can move on to compiling invoices, we have one more task: setting all values in the document itself.

3.3.1 Header

The `\makeheader` macro comes from `invoice.cls`. It expects the title and subtitle as arguments, for which we use `\param`:

```
76 \begin{document}
77 \thispagestyle{headermain}
78 \makeheader{\param{title}}{\param{subtitle}}
79 \vspace{2cm}
```

3.3.2 Information

The left column of the information is quite tricky, as it contains both client information and invoice data, such as the number and date.

```
80 \begin{tabular}{@{}l@{}}
81     \begin{tabular}{@{}l@{}}
82         \param[client]{name}\\
83         \param[client]{street}\\
84         \param[client]{postal}, ↩
             \param[client]{place}\\
85     \end{tabular} \\
86     \begin{tabular}{@{}l l@{}}
87         Invoice number: & \param{number}\\
88         Invoice date: & \param{date}\\
89     \end{tabular}
90 \end{tabular}
91 \hfill
```

You can see in the address lines that a line break is set for each line. This could also have been done if, for example, a field `address lines` of type `list` was present. Then it would have been solved in one go with `\param[client]{address lines}`, assuming that `postal` and `place` are merged on one line in YAML. This alternative assumes that the `\paramlistconjunction` macro is set to '`\\`', instead of the default '`,~`'.

```
92 \begin{tabular}{@{}l r@{}}
93     Company: & \param[supplier]{name} \\
94     Email: & \param[supplier]{email} \\
95     Website: & \param[supplier]{website} \\
96     Account nr: & ↩
         \param[supplier]{account number} \\
97     \hasparam[suplier]{extra fields}{%
98         \def\formatsupplierextra{\key & ↩
             \val\\}%
99         \fortablerow[supplier]{extra ↩
             fields}{formatsupplierextra}
100    }{}
101 \end{tabular}\\
```

The right column of information is similar to the left, except it has one additional special field, namely `extra fields` of type `table`. This allows for a variable number of rows to be added. The same could potentially be applied to the client details in the left column. Then only the choice remains whether to place them above or below the invoice information.

3.3.3 Table

As mentioned earlier, standardizing the column definition is difficult.

On line 105, you can see what the `\columdefs` could have provided, except for the counters that I previously used.

```
103 \begin{invoice}
104 % Column definition based on 540pt
105 {@{}L{180pt-\tabcolsep} ↩
      R{80pt-2\tabcolsep} ↩
      L{60pt-2\tabcolsep} ↩
      F{120pt-2\tabcolsep} ↩
      F{100pt-\tabcolsep}@{}}
```

For the second argument of the `invoice` environment, a static header is set.

```
106 % Header
107 {\textbf{Description} & \textbf{Date} & ↩
      \textbf{Quantity} & \textbf{Price} & ↩
      \textbf{Total} \\ \hline}
```

For the third argument of the `invoice` environment, you can see how the final totals are set in the table. These totals are placed in the last two columns of each row, so that they align neatly with the rest of the table.

```
% Totals
{%
    & & & \textbf{Total (ex.)} & \currency\hfill{\ttfamily \paramfield{totals}{total ex}} \\
    & & & \textbf{VAT} & \currency\hfill{\ttfamily \paramfield{totals}{vat}} \\
    & & & \textbf{Total (incl.)} & \currency\hfill{\ttfamily \paramfield{totals}{total incl}} \\
}
```

In the final part of the table, you can see how each invoice line is set using `\fortablerow` with the help of `\formatrecords`.

```
    \newcommand\formatrecords{%
        \description & \date & \quantity &%
        \currency\hfill{\ttfamily\price} &%
        \currency\hfill{\ttfamily\total} \\}
    \fortablerow{records}{formatrecords}
\end{invoice}
```

The overall structure of the table is still from the previous situation. The notable difference from the old situation is that the data can be put into any sort of table structure, since the data is decoupled from the LaTeX and application domains, and the challenges of typesetting are shifted to the LaTeX domain.

3.3.4 Closing text and images

Where we previously saw an advanced YAML specification for the `message` field, the implementation in LaTeX remains virtually the same:

```
{\footnotesize\param{message}}
```

The only difference is:

`\theending` → `\param{message}`

The images, on the other hand, are slightly more difficult to implement in LaTeX due to the `list` type.

```
\newcommand\formatimage[1] {\hspace{.75em}\includegraphics [width=2cm]{#1}\hspace{.75em}}%
\hasparam[style]{images}{%
    \vfill
    \begin{center}
        \forlistitem[style]{images} {formatimage}
    \end{center}
}{}
\end{document}
```

Where previously in Python all images were neatly placed next to each other, with a `\hspace` of `1.5em` between each image, I chose to insert half that value as an `\hspace` on each side of each image. This is because the `\forlistitem` macro does not yet have a convenient way to specify a separator, like `\param` does by setting `\paramlistconjunction` to '`\hspace{1.5em}`'.

4 Execution

Now that the legacy invoice has been completely transformed, let's see what the result looks like. If you want to participate via the command line, please refer to the full source code [4] of these examples.

4.1 The template version

Without providing any values, we get the following result, as shown in figure 7.

As mentioned earlier, `lua-placeholders` can only be compiled with LuaLaTeX. The example can be compiled as follows:

```
lualatex --jobname=invoice-template \
    --output-directory="${OUTPUT_DIR}" \
    invoice
```

Listing 15: Compiling with `lualatex`

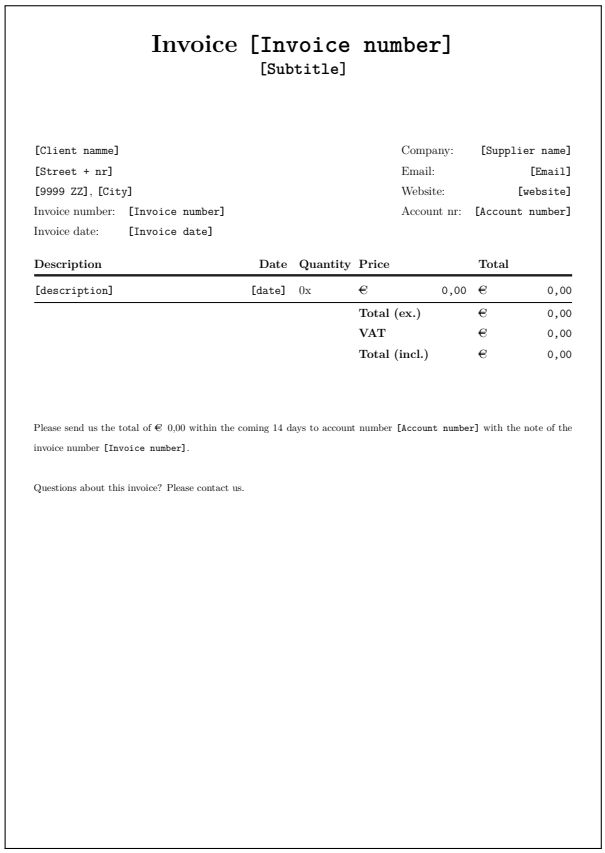


Figure 7: `invoice-template.pdf`

where `${OUTPUT_DIR}` is the desired output directory.

However, if you are designing a template, continuous generation with `latexmk` [1] is more user-friendly:

```
latexmk -pvc -lualatex \
    --jobname=invoice-template \
    --output-directory="${OUTPUT_DIR}" \
    invoice
```

Listing 16: Compiling with `latexmk`

With the `-pvc` option, you don't have to recompile with TEX every time there is a change; it happens automatically.

4.2 YAML values

To get a filled invoice, we will need the following YAML files:

⟨*project dir*⟩
- invoices
  - ⟨*invoice*⟩.yaml
- suppliers
  - ⟨*supplier*⟩.yaml
- styles
  - ⟨*style*⟩.yaml
- clients
  - ⟨*client*⟩.yaml

This structure is based on the implementation described in section 3.2.3. Before discussing the contents of the YAML files, let's first consider alternative project structures.

4.2.1 Alternative project structure

Everyone is free to create their desired folder structure. For example, you could place styles under

`/suppliers/`⟨*supplier*⟩`/style.yaml`

so that you can even omit the `style` field in the supplier recipe. Another option is to place the `clients` folder under the supplier level, so you don't accidentally mix clients of different suppliers. This could be achieved as follows:

⟨*project dir*⟩
- suppliers
  - ⟨*supplier*⟩.yaml
  - ⟨*supplier*⟩
    - ⟨*client*⟩.yaml

This way, the implementation for loading clients would require the variables ⟨*supplier*⟩ and ⟨*client*⟩, to then reach the path

`suppliers/`⟨*supplier*⟩`/`⟨*client*⟩`.yaml`.

The same consideration could be applied to the invoices, but this is a more difficult scenario, as the invoice data is based on `\jobname` in the implementation of section 3.2.3. One possible solution for this is to manage the project per supplier. You can then place the *recipes* in the `$TEXMFHOME/tex` directory so that they are available for all projects. Here's an example of a possible project structure:

`$HOME/texmf/tex`
- recipes
  - invoice.yaml
  - client.yaml
  - supplier.yaml
  - style.yaml
- invoice.cls
- invoice.tex

⟨*project dir*⟩
- invoices
  - ⟨*invoice*⟩.yaml
- clients
  - ⟨*client*⟩.yaml
- supplier.yaml
- style.yaml

In this example, all data is separated per supplier, including client information and final invoices.

4.3 Suppliers and clients

In the example result of GinVoice, a client *Juicing Joker* was shown. In YAML, this would translate to:

```
name: Juicing Joker
street: \LaTeX-street 27
postal: 12345 AB
place: Alaska
```

Listing 17: `clients/juicing-joker.yaml`

This way, the client can be referenced in the invoice with `juicing-joker`.

For the supplier, we saw *Grapefruit Inc.* in the example, which translates to:

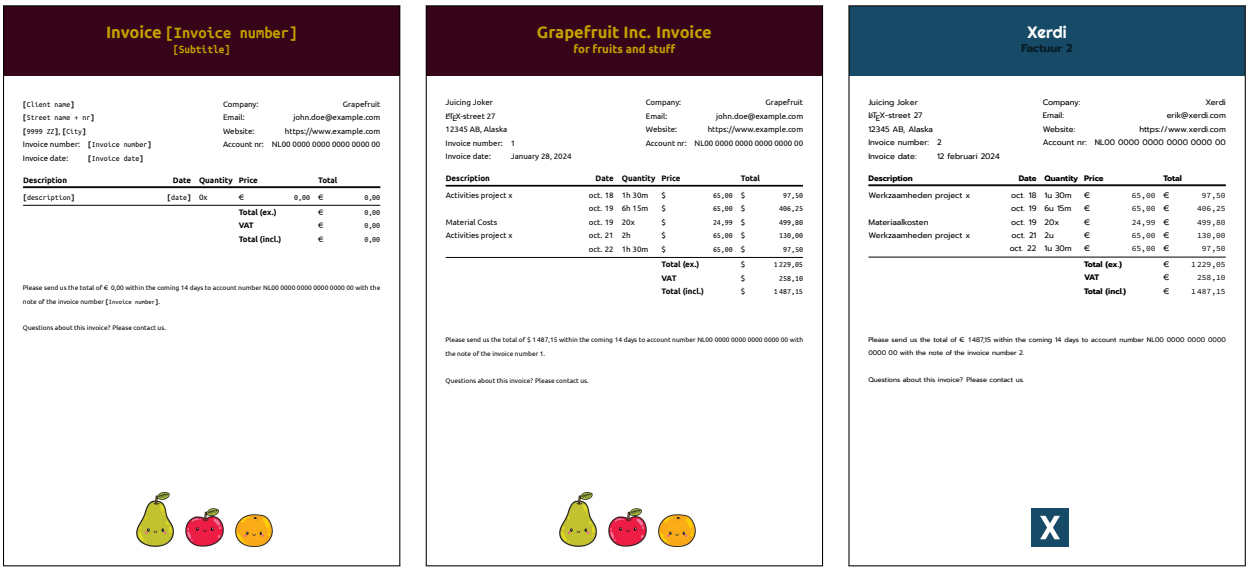
```
name: Grapefruit
email: john.doe@example.com
website: https://www.example.com
account number: NL00 0000 0000 0000 0000 00
style: grapefruit
```

Listing 18: `suppliers/grapefruit.yaml` or `grapefruit/supplier.yaml`

And for the style:

```
main font: Ubuntu
mono font: Ubuntu Mono
foreground color: c4a000
background color: 360519
images:
  - img/image1
  - img/image2
  - img/image3
```

Listing 19: `styles/grapefruit.yaml` or `grapefruit/style.yaml`



(a) `invoice-template.pdf` (b) `invoice-001.pdf` (c) `invoice-002.pdf`

Figure 8: Invoice Examples

The advantage of the alternative project structure is that `invoice-template` automatically picks up the styling as well as the supplier information, as seen in figure 8a.

4.4 Invoices

To create an invoice that exactly matches the standard example of GinVoice, as seen in figure 8b, we use the following YAML example:

```yaml
supplier: grapefruit
client: juicing-joker
title: Grapefruit Inc. Invoice
subtitle: for fruits and stuff
currency: \$
number: 1
date: January 28, 2024
records:
  - description: Activities project x
    date: oct. 18
    quantity: 1h 30m
    price: 65
    total: 97.5
  - description: ''
    date: oct. 19
```

Listing 20: `invoices/invoice-001.yaml`

The actors `grapefruit` and `juicing-joker`, discussed in section 4.3, are seen in the invoice. Additionally, the example has the same general information to achieve the same result. In the `records` field, you can see that one row of the table takes up many lines. In the second row of the table, you can see that the `description` field has an empty value. If the quotes are omitted in YAML, you will get an error when converting to data. Since the rows do not differ too much from each other, we continue the example at the `totals` field:

```yaml
totals:
  total ex: 1229.05
  vat: 258.10
  total incl: 1487.15
message: |
  Please send us the total of \currency~\paramfield{totals}{total incl}
  within the coming 14 days to account number
  \param[supplier]{account number} with the note of the invoice number \param{number}.\\[2em]

  Questions about this invoice? Please contact us.
```

Lastly in the example, we see the totals and the closing text.

This invoice can then be compiled with the following command:

```
lualatex --jobname=invoice-001 \
    --output-directory="${OUTPUT_DIR}" \
    invoice
```

5 Conclusion

In this study, we have not only examined the implementation of invoice templates in GinVoice but also proposed an innovative method to seamlessly integrate these templates with the LaTeX ecosystem. By using YAML as an intermediate layer and `lua-placeholders` for dynamic insertions, we have provided a robust and flexible solution for invoice generation while creating a framework where various document components, such as client information, can be utilized across documents.

This approach not only grants LaTeX users the freedom to customize invoice templates as desired but also opens the door to a wider range of applications. By employing the same YAML-based structure, different documents, including contracts and invoices, can be generated and maintained with ease. This not only enhances consistency across various document types but also boosts the efficiency of the documentation process as a whole.

The utilization of `lua-placeholders` in conjunction with YAML enables the addition of dynamic content to templates, resulting in a more streamlined workflow for users. This flexibility makes it easy to separate data and formatting across different documents while allowing these components to be used across documents.

In conclusion, this approach not only makes a valuable contribution to optimizing billing processes but also unveils new possibilities for efficiently generating and managing various types of documents within an organization.

6 Discussion

6.1 LaTeX compilers

In the article, I assume the LuaLaTeX compiler. For other compilers, `lua-placeholders` does not provide a solution. Although some compilers still offer support for Lua, `lua-placeholders` does not take this into account. Research and implementation could improve the adoption of `lua-placeholders` within the LaTeX community.

6.2 JSON vs. YAML

I did not delve into the choice of YAML over JSON in the article. Both are intended for data, and while JSON is more well-known and has broader compatibility with programming languages, I chose YAML for the sake of readability of LaTeX source code. As demonstrated extensively, the files contain a lot of LaTeX source code. When using JSON every backslash would need to be escaped. For example:

```
title: Invoice \param{number}
```

Listing 21: YAML example

```
{"title": "\\param{number}" }
```

Listing 22: JSON example

As a LaTeX user, I find it more convenient to adjust values in YAML for testing purposes than in JSON.

6.3 GinVoice roadmap

Development has been stagnant for some time, but I recently discovered that the solution can also work for Windows platforms. Bringing GinVoice to the Windows platform significantly expands the target audience and, in my expectation, could garner more support for LaTeX.

As for the introduction of `lua-placeholders`, there are still a few obstacles to overcome, such as challenges related to translation and the variable column definition, which is precisely a user-friendly part of the application that has not been discussed.

References

[1] J. Collins, E. McLean, D.J. Musliner. *The latexmk package.* `www.cantab.net/users/johncollins/latexmk/index.html`

[2] S. Dierl. *The invoice2 package.* `github.com/no-preserve-root/invoice2`

[3] E. Nijenhuis. *The GinVoice GTK application.* `gitlab.gnome.org/MacLotsen/ginvoice`

[4] E. Nijenhuis. *The GinVoice template.* `github.com/Xerdi/ginvoice-template/tree/maps`

[5] E. Nijenhuis. *The lua-placeholders package.* `ctan.org/pkg/lua-placeholders`

◇ Erik Nijenhuis
Frans Halsstraat 38
Leeuwarden, 8932 JC
The Netherlands
`erik (at) xerdi dot com`
`https://github.com/MacLotsen`