Minimum(ish) Working Example of Quarto with Arabic RTL text in an LTR document

The Authors

2024-11-16

This book is typeset with XeLaTeX (via Quarto and pandoc) in the Charis, Vazirmatn, and Amiri typefaces.

Table of contents

Preface

بسم الله الرحمن الرحيم

Quarto is a document publishing software. With it, you can write your document in Pandoc flavored markdown. Quarto will use Pandoc under the hood, and do a bunch of other fancy stuff, to output your markdown document in formats of your choice, like HTML for websites, and PDF (via Latex).

So far so good. But many of my documents are in English with Arabic content interspersed. Arabic is written right-to-left (RTL) whereas English is written left-to-right (LTR). The support of bidirectional (BiDi) text is a notoriously tricky problem.

In this write-up, I will describe how to configure Quarto to achieve this.

The code for this book can be used as a template for RTL document projects. Along with BiDi, I'll also discuss other aspects like fonts, figures, etc.

The source code for this book can be found here: https://github.com/adamiturabi/quarto-arabic-mwe.

The rendered output is published here: https://adamiturabi.github.io/quarto-arabic-mwe.

1 Introduction

In order to create a Quarto book project (like this one), use this command:

```
quarto create project book mybook
```

It will create a bunch of files that will be needed for the book.

This procedure is described in detail here: https://quarto.org/docs/books/.

2 Arabic (العربية) support

There are two main ways to insert RTL Arabic text in an LTR document:

1. In-line Arabic within an LTR paragraph.
2. A block of Arabic text by itself

2.1 In-line Arabic: spans

For inline Arabic, we will use a Pandoc span. A span is written using this syntax:

```
[This is the span's *content text*]{.classname attributekey="attributeval"}
```

Within square brackets `[]` is the content of the span. This is what will be randered in the output. Within the curly braces `{}` is a class name and some attributes that are needed by Quarto to properly process the span.

In order to render the Arabic content text correctly for HTML output, the span is typed thus in the `.qmd` source file.

```
[هذا نص عربي.]{.reg-ar-txt dir="rtl" lang="ar"}
```

The class name is arbitrary. We suggest a descriptive name. We will be using it in the CSS for selecting the font later. The output HTML code will be something like:

```
<span class="reg-ar-txt" dir="rtl" lang="ar">هذا نص عربي.</span>
```

For PDF output, the `dir="rtl"` attribute is unneeded and actually clashes with the Xelatex PDF engine that Quarto mandates we use for documents with RTL text. So the span will need to be typed thus in the `.qmd` source file:

```
[هذا نص reg.}[.عربي-ar-txt lang="ar"}
```

The output Latex code will be something like:

```
\foreignlanguage{arabic}{هذا نص {.عربي
```

Under the hood, `\foreignlanguage` is a command that is used by the Latex package `babel` that Pandoc specifies in its Latex template for handling multiple languages and their scripts.

Finally, this is an example of an English sentence with inline Arabic text نَصٌّ عَرَبِيٌّ. within it. Locate this sentence in the source code file here to see how we wrote it.

2.2 Arabic block text: divs

In order to write a block (one or more paragraphs) of Arabic text within an LTR document we will use a Pandoc div. A div is written using this syntax:

```
:::{.classname attributekey="attributeval"}
This is the divs's *content text*.

It may comprise multiple lines. It may contain blank lines.
:::
```

For HTML output, a div is typed thus in the `.qmd` source:

```
:::{.reg-ar-txt dir="rtl" lang="ar"}
نص عربي.

نص عربي مزيد.
:::
```

The class name `reg-ar-txt` is, again, arbitrary. The output HTML code will be:

```
<div class="alt-ar-text" lang="ar" dir="rtl">
هذا نص عربي.

هذا نص عربي مزيد.
</div>
```

For PDF output, a div is typed thus in the `.qmd` source:

```
:::{.otherlanguage data-latex="{arabic}" lang='ar'}
هذا نص عربي.

هذا نص عربي مزيد.
:::
```

In this case, the class name `otherlanguage` is not arbitrary. Furthermore another attribute `data-latex="{arabic}"` is also needed. And as with spans `lang="ar"` is needed but `dir="rtl"` should not be used. The output Latex code is:

```
\begin{otherlanguage}{arabic}

\altfamily

هذا نص عربي.

هذا نص عربي مزيد.

\end{otherlanguage}
```

Finally, this is an example of an Arabic div. Locate it in the source code file here to see how we wrote it.

هذا كلام عربي طويل. أريد أن أكتب حتى يبلغ الكلام سطرين. أستعمل برنامج قواطرو لإنتاج الملف الخارجي. هو برنامج جيد قد خلف البرنامج بكداؤن الذي كنت أستعمله من قبل.

2.3 Pandoc Lua fiters

As you can see, the process for typing Arabic text is both lengthy, and different for HTML and PDF outputs. In order to simplify it, we can use Pandoc Lua filters.

We have created a Quarto filter extension (which is a grouping of Lua filters) to support Arabic divs and spans. The process for creating a Quarto filter extension is detailed here: https://quarto.org/docs/extensions/filters.html

This is the filter `inline-arabic-span.lua` that we wrote for handling Arabic spans:

```
-- Add attributes for Arabic text in a span
function Span (el)
 if el.classes:includes 'ar' or el.classes:includes 'aralt' then
    text = pandoc.utils.stringify(el)
    contents = {pandoc.Str(text)}
    if FORMAT:match 'latex' then
      -- for handling alternate Arabic font
      if el.classes:includes 'aralt' then
     -- can't seem to use string concatenate directly. Have to use RawInline
        table.insert(
          contents, 1,
          pandoc.RawInline('latex', '\\altfamily ')
        )
      end
    -- no dir needed for babel and throws error if it sees dir attribute. was previo
      return pandoc.Span(contents, {lang='ar'})
    elseif FORMAT:match 'html' then
      classval = 'reg-ar-text'
      if el.classes:includes 'aralt' then
       classval = 'alt-ar-text'
      end
     -- dir needed for html otherwise punctuation gets messed up
    return pandoc.Span(contents, {class=classval, lang='ar', dir='rtl'})
    end
  end
end
```

This is the filter `arabic-div.lua` that we wrote for handling Arabic divs:

```
-- Add attributes for Arabic text in a div
function Div (el)
 if el.classes:includes 'ar' or el.classes:includes 'aralt' then
    text = pandoc.utils.stringify(el)
    contents = {pandoc.Str(text)}
    if FORMAT:match 'latex' then
      -- for handling alternate Arabic font
      if el.classes:includes 'aralt' then
     -- can't seem to use string concatenate directly. Have to use RawInline
        table.insert(
          contents, 1,
          pandoc.RawInline('latex', '\\altfamily ')
        )
      end
    -- no dir needed for babel and throws error if it sees dir attribute. was p
    return pandoc.Div(contents, {class='otherlanguage', data_latex="{arabic
    elseif FORMAT:match 'html' then
      classval = 'reg-ar-text'
      if el.classes:includes 'aralt' then
        classval = 'alt-ar-text'
      end
     -- dir needed for html otherwise punctuation gets messed up
    return pandoc.Div(contents, {class=classval, lang='ar', dir='rtl'})
    end
  end
end
```

With activating these filters, now you can use Arabic divs and spans using a simplified syntax.

Input for an Arabic span:

```
[هذا نص {ar.}][.عربي
```

Input for an Arabic div:

```
:::{.ar}
```

```
نص .عربي

نص عربي .مزيد
:::
```

The filters will process them correctly for HTML and PDF output. Note that the class name `reg-ar-text` is hardcoded in the filter. If you wish to modify it you can edit the Lua files directly.

2.4 Arabic fonts

You can use a specific font for the Arabic text which is different from the font used for the English text. This is usually desirable because the typeface design for the Latin font often does not optimize (or even sometimes support) an Arabic font.

For my project, I am using the Vazirmatn and Amiri fonts.

Both of these are well designed fonts. For me a major consideration is the correct typesetting of the hamza character (ء). See here for what I'm talking about: https://adamiturabi.github.io/hamza-rules/

To specify the Arabic fonts the process is different for HTML vs PDF output. We'll describe both below:

2.4.1 Specify Arabic font for HTML

For HTML output, the Arabic font is specified in the CSS file. The class name that we selected previously `reg-ar-text` is now assigned a font:

```
.reg-ar-text {
  font-family: Vazirmatn, serif;
 /* scaled up slightly w.r.t. the Latin font for readability */
  font-size: 1.2em;
 /* line spacing not scaled for visual congruence at the expense of clashes */
  line-height: 100%;
}
.alt-ar-text {
```

```
  font-family: AmiriWeb, serif;
  font-size: 1.2em;
  line-height: 100%;
}
```

The font names `Vazirmatn` and `AmiriWeb` are specified in the same CSS file. See our CSS file for details.

Amiri is specified as an alternate font. It can be specified in the `.qmd` source with `{.aralt}` instead of `{.ar}`. You can also see how we handle it in the source code for the Lua filters above.

By the way, I am, by no means, at expert (or even proficient) in CSS, so if you see any problems with this method, feel free to let me know in the discussions or issues of the Github page for this book.

2.4.2 Specify Arabic font for PDF

As we mentioned earlier, Latex uses the babel package to handle multiple languages. In order to specify Arabic font(s), we need to add the following lines in the intermediate `.tex` file produced by Quarto:

```
\babelfont[arabic]{rm}[Language=Default]{Vazirmatn-Light}
\babelfont[arabic]{sf}[Language=Default]{Vazirmatn-Light}
\babelfont[arabic]{alt}[Language=Default]{Amiri}
```

Quarto provides hooks for inserting such additional code using includes and templates.

The above lines of code need to be inserted at a specific point after the `usepackage{babel}` line. We found that replacing the partial template for `before-title.tex` worked in this case. Here is the addition in our `_quarto.yml` file:

```
format:
  pdf:
    template-partials:
      - srctex/before-title.tex
```

Again, see our source code in Github for more details.

3 Transliteration of Arabic

In my work, I frequently need to transliterate and transcribe Arabic text in Latin characters. There are various Romanization schemes in existence, using dots, macrons, etc. The Romanization scheme I am using for my work is tabulated below:

Arabic letter	Transliterated output	ASCII input
ء	ʾ	E
ا	ā	A
ب	b	b
ت	t	t
ث	t͡h	v
ج	j	j
ح	ḥ	H
خ	k͡h	x
د	d	d
ذ	d͡h	p
ر	r	r
ز	z	z
س	s	s
ش	s͡h	c
ص	ṣ	S
ض	ḍ	D
ط	ṭ	T
ظ	ḍ͡h	P
ع	ɛ	e
غ	g͡h	g
ف	f	f
ق	q	q
ك	k	k
ل	l	l
م	m	m

Arabic letter	Transliterated output	ASCII input
ن	n	n
ه	h	h
و (C)	w	w
ي (C)	y	y
و (V)	ū	U
ي (V)	ī	I

As you can see, I use digraphs *d͡h*, *g͡h*, etc. for some letters. This is because for my current work, I value readability over precision.

It is possible to input these special characters directly by modifying your keyboard layout or mapping, either at a operating system, or editor level. Andreas Hallberg has described a technique for inputing them in the vim editor here: https://andreasmhallberg.github.io/ergonomic-arabic-transcription/

For Quarto, I prefer to input the transliterated text as ASCII characters and write a Lua filter `transliteration-span.lua` to handle rendering them correctly. The ASCII to transliterated output mapping is shown in the table above. So if I type:

```
[pahabtu maphaban]{.trn}
```

It will be output as *d͡hahabtu madh͡h·haban*.

Note the dot character · is automatically inserted by the filter between the digraph *d͡h* and the following *h* for helping in disambiguation.

With `{.trn}` the output is in italic (as above). But sometimes I need to have non-italic output, as in the case of names. For that I use `{.trn2}`. For example:

```
[#eumar]{.trn2} and [E#usAmah]{.trn2} are studying
the [#qurEAn]{.trn2} and [#HadIv]{.trn2}.
```

This is rendered as:

Ɛumar and ʾUsāmah are studying the Qurʾān and Ḥadīt͡h.

Note how the hash character # is used to control capitalization.

For the English font used in your main text, you will need to pick a font that supports the dots, macrons, breves, etc needed for transliteration. I am using the Charis SIL font.

4 Summary

Dummy text to test references:

See Knuth (1984) for additional discussion of literate programming.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.