Devdoc-Swissknife EN

Nikolay Gniteev

2022-05-14

Contents

List of Tables

List of Figures

Preface

This is a demo project for development documentation generation with **R Markdown** and **Kroki**.

All the things described here you could do without that project, but I've tried to stich those components together so you'll be able to concentrate on content rather than process.

This approach allows you to create reproducible documentation in form of PDF, presentations, gitbook and some other HTML formats from simple markdown-like files (Pandoc flavor) with extra **R chunks** and textual diagrams description.

The goal of this project is to describe documentation via text files as much as possible, even graphic diagrams, and make it easy to use without overcomplication.

A textual description provides an easy way to version and merge diagrams, get differences between versions, leave comments that stays out from generated documentation, review changes, etc.

Also most often it's faster to create and **edit** diagrams as text, do things like theming and refactoring themes. Since the diagram's textual description often is more a model than just graphics, you don't have to track existing objects relations when just placement is changed or objects added (until you rename objects) which makes your workflow less error-prone.

Embedding textually described diagrams into docuement is the actual focus of this porject. The rest you would be able to do with R Markdown itself as it is a powerful and customizable tool, so you can fit docs to your needs. It would take efforts on start, to make things neat but it would repay you in the end.

> To produce awesome PDFs you'll have to provide awesome Latex book class. Since I don't have one I'm using Latex default so current PDF version looks a bit ugly to me. I hope I can fix this soon or later. Help is appreciated!

Kroki is running locally in container so you may not worry for confidential things and it works well in air gapped installations.

This project is also can be used as a template for your own documentation, see this section

The project uses docker and docker-compose to build docs so it's ready to be integrated into your CI/CD flow.

The project's structure, docs generation and some files are based on R Markdown book

---

There are few demonstrations in the project for general development use-cases (which probably cover 95% of your daily needs). For more complicated use-cases - checkout docs on links below.

> Diagrams code for most examples is borrowed straight from Kroki

---

Usage section would be updated eventually with reciepes for common cases.

Helpful links

Diagrams

List of supported diagrams renderers

A few more diagrams examples

R Markdown

R Markdown: The Definitive Guide

R Markdown: Cookbook

Bookdown: Authoring Books and Technical Docuements with R Markdown

R Markdown Site (focused on usage with **R Studio**)

A nice R Markdown tutorial

R Markdown reference

Get quick into R Markdown

R Makedown anatomy

R Markdown short reference to get familiar with most necessary things

R Markdown cheatsheet

Editors with focus on diagrams

Keenwrite - edit and preview R Markdown Live!

R Studio

Alternatives

https://bookdown.org/yihui/rmarkdown-cookbook/diagrams.html / https://rich-iannone.github.io/DiagrammeR/

Chapter 1

Simple diagrams examples

Since this project is focused on embedding diagrams textual descriptions into **R Markdown**, let's start with few examples just to illustrate a principle.

> BTW: there is a lot of drawings types that you can get from **R Markdown** alone, by using **R** packages or Python's matplot lib for example. You may want to check this out too.

Example with inline diagram description

This example shows how to embed diagrams, describing them right in the `.Rmd` files.

~~~
```{r, hello-world, dia="graphviz", fig.cap="Hello World"}
  to_diagram("digraph G {Hello->World}")
```
~~~

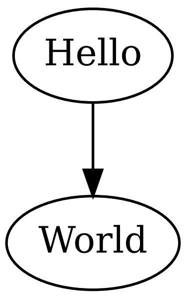


Figure 1.1: Hello World

Example with diagram data from file

This example shows how to embed diagrams with data from outer files.

~~~
```{r, examples-entity-relation, dia="erd", fig.cap="Entity Relation"}
  to_diagram(src="../diagrams/examples/project.erd")
```
~~~

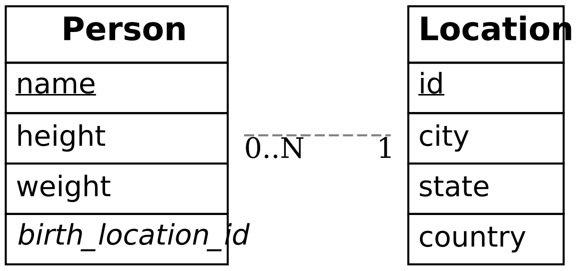


Figure 1.2: Entity Relation

File **diagrams/examples/project.erd** content:

```
[Person]
*name
height
weight
+birth_location_id

[Location]
*id
city
state
country

Person *--1 Location
```

More examples

More examples for your inspiration can be found in this section.

Details about embedding diagrams is here

Chapter 2

Using as a template and making own docs

To use this project as a start point for generating your own documentation do the following:

1. Import project and prepare for making docs

   You'll need to have already `docker` and `docker-compose` installed

- Import or fork main branch of this repo: https://github.com/Godhart/devdoc-swissknife.
  All the sources for docs are contained in `docs_src` folder.
- Make docker image `devdoc-swissknife` if you don't have one already.
  Simply run `make_docker.sh` from `docker` dir.
- Ensure that everything works fine - try to make docs with `make_docs.sh`.
  Output docs should appear in `docs_out/devdoc-swissknife-*` folders. Check that docs exist and contain graphics.
  Take a NOTE: `docs_out` folder and all it's content is ignored by git.
- If you don't want to see devdoc-swissknife documentation in your repo:
  - Remove `docs_src/devdoc-swissknife-*` folders
  - Remove `docs_out/devdoc-swissknife-*` folders

- Empty `docs_src/diagrams` folder.

2. Prepare for making your own docs

- Make a new folder in `docs_src`. I suggest you name it using the following pattern: `doc-<subject>`
- Copy all the files from `docs_src/docs-template` into your brand new folder
- Replace following keywords `<Author Name>`, `<author>`, `<repo>`, `<Document Title>`, `Document_Title`, `<Document Description>`, `doc-` within files in your new folder with actual values. Also don't forget to adjust `Document_Title` in the `.gitignore` of new folder, as it protects from build garbage.
- Adjust `docs_src/Makefile` to fit your needs (for the first time - use an existing pattern to add your folder).
- Ensure that everything is still fine - try to make docs with `make_docs.sh`.
  Output docs should appear in `docs_out/doc-<subject>` folder if you followed a pattern in the Makefile.

  NOTE: in many cases of errors doc's sources folder (`docs_src/doc-<subject>` if you follow suggested pattern) is polluted with temporary files, named as specified in `_bookdown.yml` (field `book_filename`) and may break following docs generation runs.
  These files are removed with make routine, but in some cases, you'll have to remove them by yourself.
  Changing `book_filename` field in `_bookdown.yml` after an error has happened may be the case.

3. Create content

- Update `index.Rmd` in your folder to your needs (contains Preface section).
- Add your own docs into your folder, naming files like `<number>-<chapter-name>.Rmd`.
  Check the **R Markdown** and **Kroki** docs for understanding things. Also, you may rely on shown examples.
- If you already have docs in markdown format you may already use them like this:

  - copy markdown files into your doc-folder
  - copy necessary local images to location in `docs_src/diagrams` or wherever you like most
  - change name extensions of markdown files to `.Rmd`
  - change names of markdown files so they would correspond to pattern `<number>-<chapter-name>.Rmd`
  - change references to local images in markdown files

- If you already have text description of diagrams for supported rendering engines you may already use them like this:

  - copy necessary files to location in `docs_src/diagrams` or wherever you like most
  - in `.Rmd` files replace image embedding with diagram embedding as described TODO

- Most probably you would like to use your own Latex class, so add `<your_latex_class>.cls` file into your folder and specify it in `index.Rmd` file (replace `documentclass: book` field with `<your_latex_class>` name).

  Adding custom latex class may require you to add some latex packages to Docker image. Same is true if you do use some special **R** packages in your docs etc.
  If this is the case you'll have to modify `docker/Dockerfile` and build again docker image with `make_docker.sh`.

Chapter 3

Kroki usage examples

I've omitted some examples from Kroki and left only those that are most useful in a daily life of most developers (IMO).

If you check docs for supported diagrams renderers then you'll find for sure a few more interesting usecases.

All diagrams data for examples of this section resides in docs_src/diagrams/examples dir of this repo. Each diagram is included into doc by adding following code section into document file:

~~~
```{r, <reference-label>, dia="from_src", fig.cap="<Caption for your figure>"}
  to_diagram(src="../diagrams/<src_file_path_within_diagrams_dir>")
```
~~~

Full code for this whole section is here

This usage pattern is described in the following section TODO

3.1 C4 Context Diagram (PlantUML+C4)

Engine: `c4plantuml`

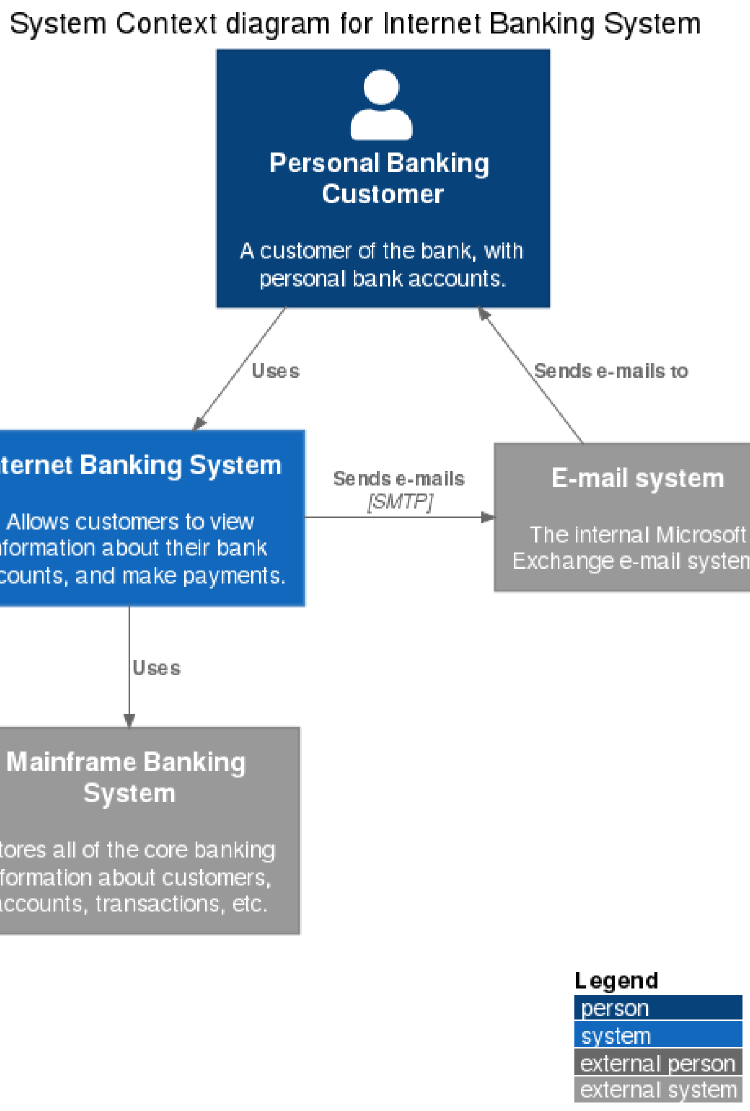

System Context diagram for Internet Banking System
Personal Banking Customer
A customer of the bank, with personal bank accounts.
Uses
Sends e-mails to
Internet Banking System
Allows customers to view information about their bank accounts, and make payments.
Sends e-mails
[SMTP]
E-mail system
The internal Microsoft Exchange e-mail system.
Uses
Mainframe Banking System
Stores all of the core banking information about customers, accounts, transactions, etc.
Legend
person
system
external person
external system

Figure 3.1: Example - C4 Context Diagram

*NOTE: inserted as PNG image due to errors in SVG to PDF conversion

3.2 C4 Container Diagram (PlantUML+C4)

Engine: `c4plantuml`

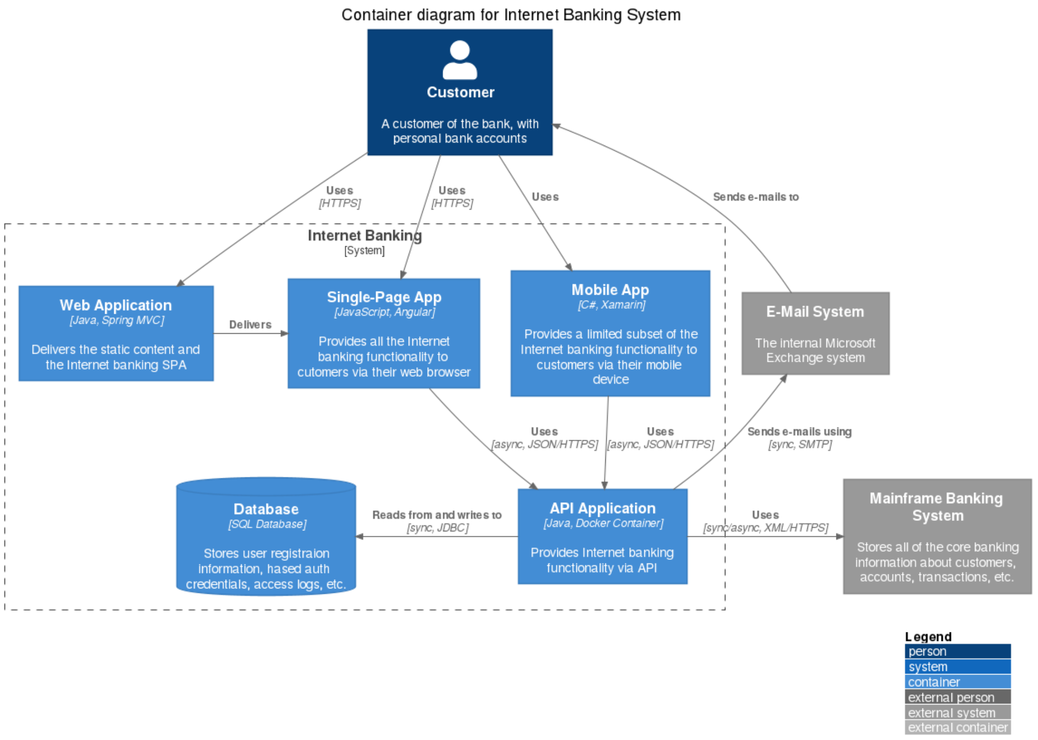

Container diagram for Internet Banking System
Customer
A customer of the bank, with personal bank accounts
Uses [HTTPS]
Uses [HTTPS]
Uses
Sends e-mails to
Internet Banking [System]
Web Application [Java, Spring MVC]
Delivers the static content and the Internet banking SPA
Delivers
Single-Page App [JavaScript, Angular]
Provides all the Internet banking functionality to cutomers via their web browser
Mobile App [C#, Xamarin]
Provides a limited subset of the Internet banking functionality to customers via their mobile device
E-Mail System
The internal Microsoft Exchange system
Uses [async, JSON/HTTPS]
Uses [async, JSON/HTTPS]
Sends e-mails using [sync, SMTP]
Database [SQL Database]
Stores user registraion information, hased auth credentials, access logs, etc.
Reads from and writes to [sync, JDBC]
API Application [Java, Docker Container]
Provides Internet banking functionality via API
Uses [sync/async, XML/HTTPS]
Mainframe Banking System
Stores all of the core banking information about customers, accounts, transactions, etc.
Legend
person
system
container
external person
external system
external container

Figure 3.2: Example - C4 Container Diagram

*NOTE: inserted as PNG image due to errors in SVG to PDF conversion

3.3 C4 Component Diagram (PlantUML+C4)

Engine: `c4plantuml`

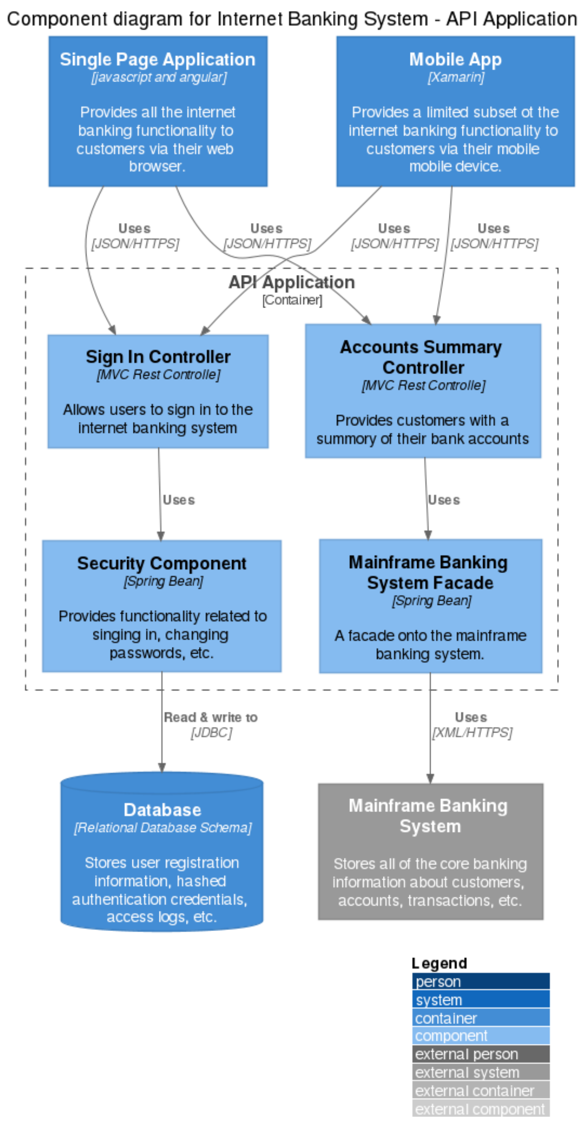

Component diagram for Internet Banking System - API Application
Single Page Application
[javascript and angular]
Provides all the internet banking functionality to customers via their web browser.
Mobile App
[Xamarin]
Provides a limited subset of the internet banking functionality to customers via their mobile mobile device.
Uses
[JSON/HTTPS]
Uses
[JSON/HTTPS]
Uses
[JSON/HTTPS]
Uses
[JSON/HTTPS]
API Application
[Container]
Sign In Controller
[MVC Rest Controlle]
Allows users to sign in to the internet banking system
Accounts Summary Controller
[MVC Rest Controlle]
Provides customers with a summory of their bank accounts
Uses
Uses
Security Component
[Spring Bean]
Provides functionality related to singing in, changing passwords, etc.
Mainframe Banking System Facade
[Spring Bean]
A facade onto the mainframe banking system.
Read & write to
[JDBC]
Uses
[XML/HTTPS]
Database
[Relational Database Schema]
Stores user registration information, hashed authentication credentials, access logs, etc.
Mainframe Banking System
Stores all of the core banking information about customers, accounts, transactions, etc.
Legend
person
system
container
component
external person
external system
external container
external component

Figure 3.3: Example - C4 Component Diagram

*NOTE: inserted as PNG image due to errors in SVG to PDF conversion

3.4 Block Diagram

Engine: `blockdiag`

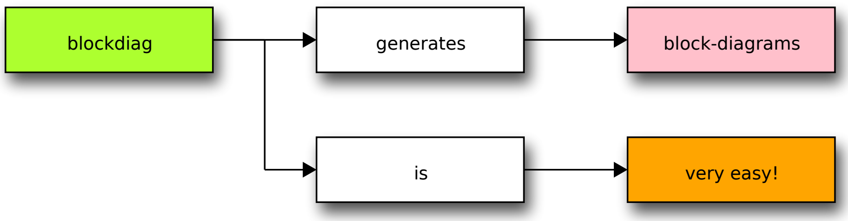


Figure 3.4: Example - Block Diagram

3.5 Digital Timing Diagram

Engine: `wavedrom`

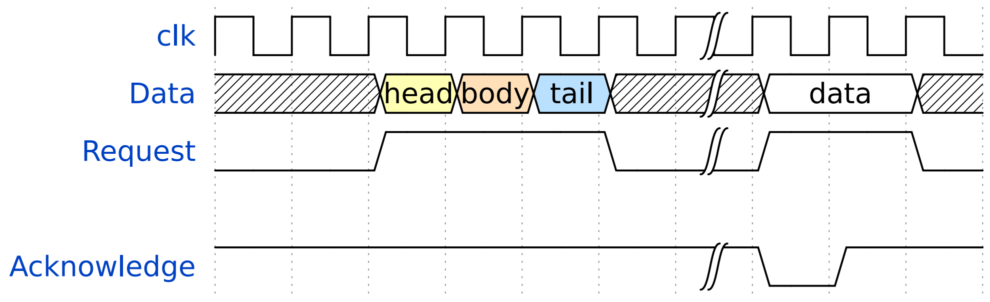


Figure 3.5: Example - Digital Timing Diagram

3.6 Bytefield

Engine: `bytefield`

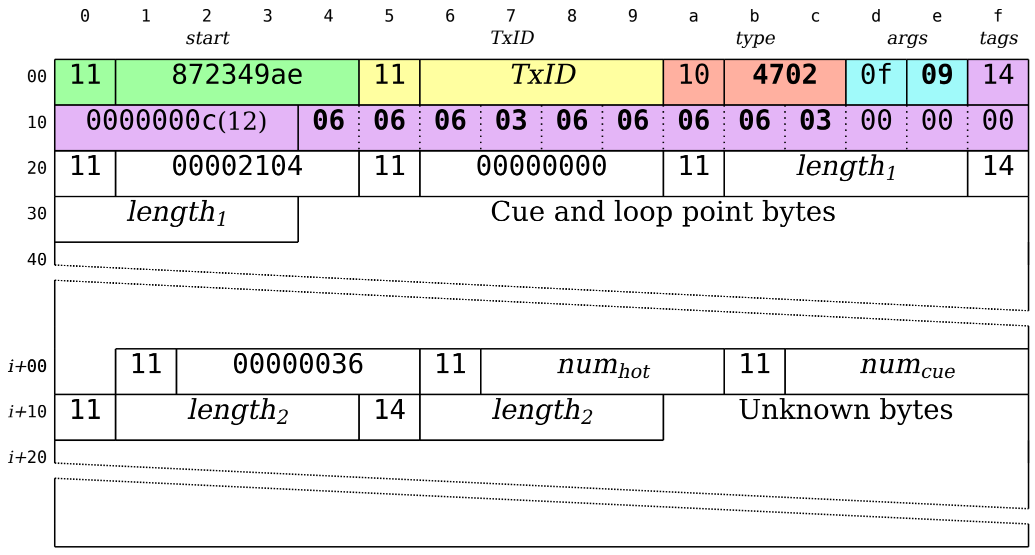


Figure 3.6: Example - Bytefield

3.7 Packet Diagram

Engine: `packetdiag`

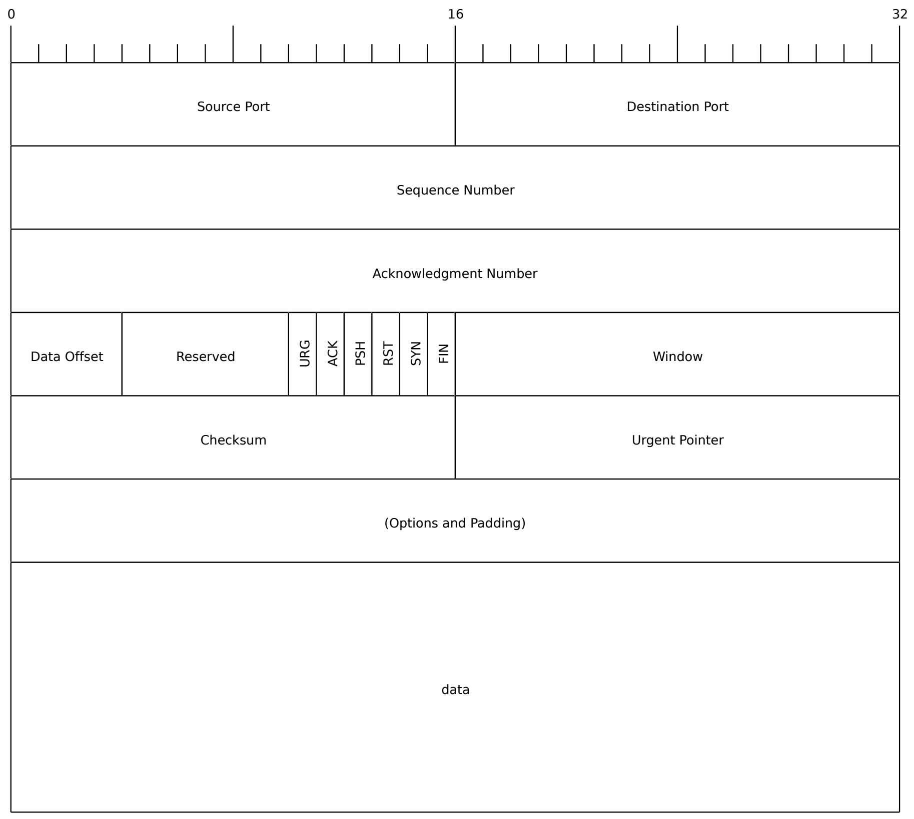


Figure 3.7: Example - Packet Diagram

3.8 Sequence Diagram #1 (PlantUML)

Engine: `plantuml`

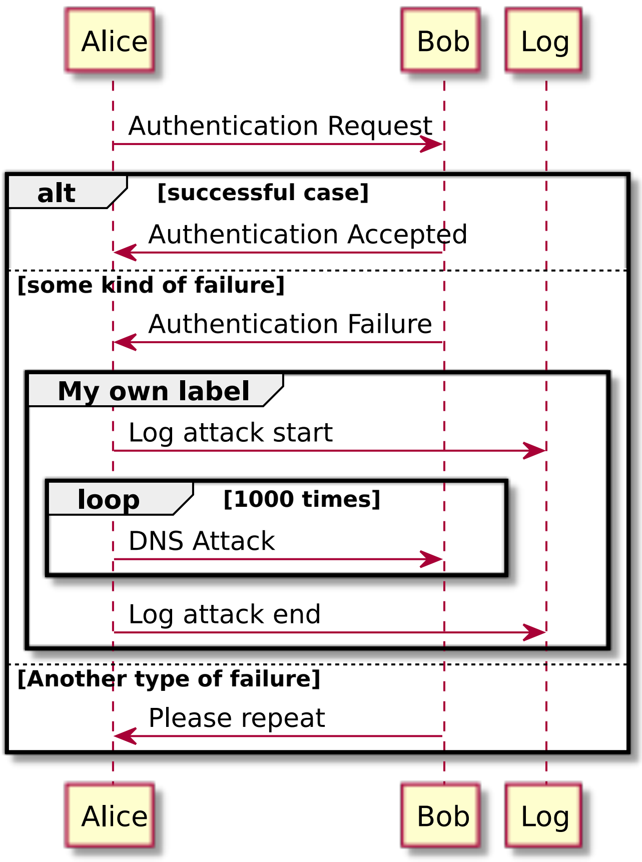


Figure 3.8: Example - Sequence Diagram - PlantUML

3.9 Sequence Diagram #2 (SeqDiag)

Engine: `seqdiag`

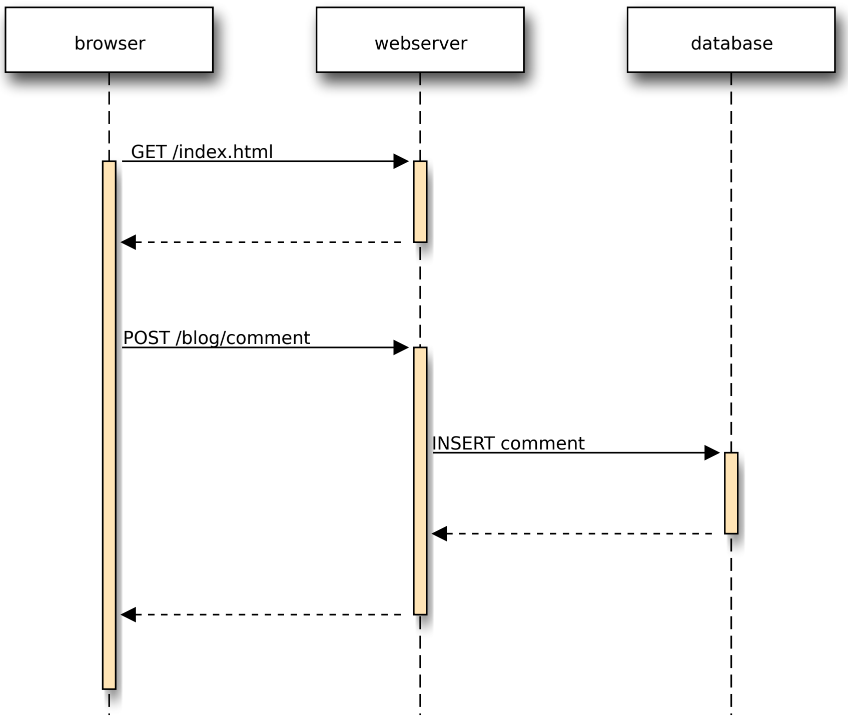


Figure 3.9: Example - Sequence Diagram - SeqDiag

3.10 Commit Graph

Engine: `pikchr`

> NOTE: pikchr is giving troubles in PDF/PNG (produced
> SVG output is only web-browser friendly)
> That’s why this diagram is absent in PDF version

3.11 Use Case Diagram

Engine: `plantuml`

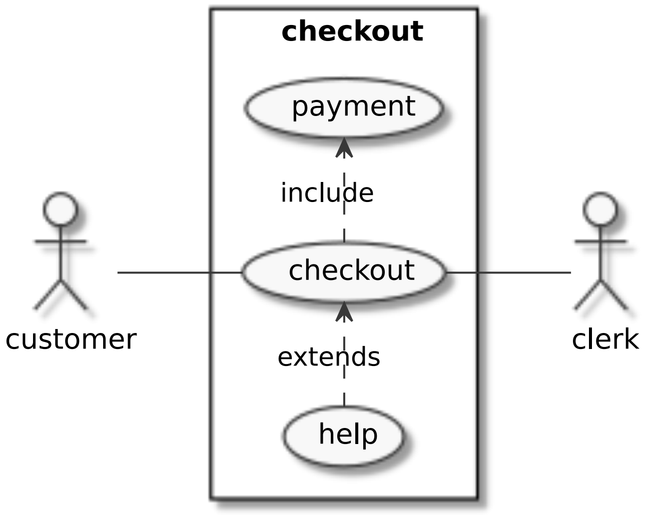


Figure 3.10: Example Block Diagram

3.12 Mind Map

Engine: `plantuml`

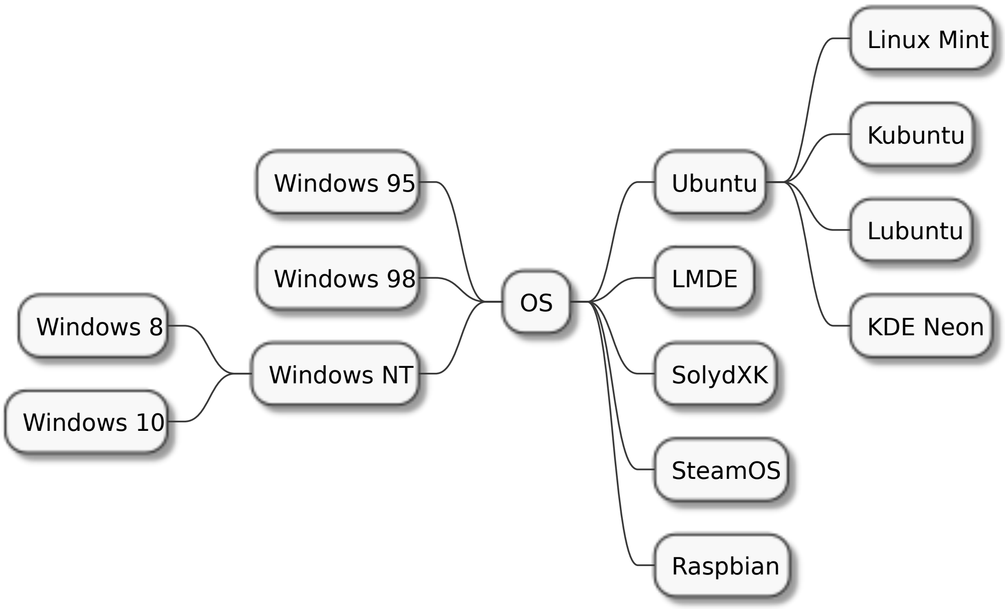


Figure 3.11: Example - Mind Map

3.13 PlantUML (More examples)

PlantUML supports more diagram types like timing diagram, gantt and many more.

You can use any of them just like in previous examples.

Check PlantUML docs https://plantuml.com/ for filling in diagram data.

3.14 Gantt

Engine: `mermaid`

> NOTE: mermaid is giving troubles in PDF/PNG (produced SVG output is only web-browser friendly) That's why this diagram is absent in PDF version

Chapter 4

Usage patterns

There are few proposed ways to embed diagrams into a document and to customize them.

Which one to use depends on circumstances.

> This section would be updated as project evolves

4.1 Embedding diagrams into a document

Let's cover basics on embedding diagrams into documentation first.

4.1.1 Describing diagrams inline in the documentation

Useful for short and/or strongly context-related drawings, i.e. when drawing by itself have no meaning or there is no reason for storing it in a separate file.

Following pattern should be used:

~~~
```{r <label>, dia='<engine>', fig.cap='<drawing name>'}
  to_diagram('<diagram data>')
```
~~~

where:

- `<label>` is a name for a drawing reference, may be omitted
- `<engine>` is a name for a rendering engine, see table for allowed values and corresponding engines
- `<drawing name>` is a caption for drawing, may be omitted.
- `<diagram data>` is a textual diagram description in a format that is compatible with a rendering engine
  NOTE: if there are single quotes in the `<diagram data>` then you should escape them with `\` symbol or surround `<diagram data>` with double-quotes (`"`) instead.

4.1.2 Diagrams from raw outer files

This is the case if you already have a diagram description in a separate file and you want to keep it like that for some reason.

> For new diagrams it's proposed to use method, described in the next section. It also uses outer files, but requires some extra formating.

Raw file is a text file that contains diagram description data in a format that is compatible with a rendering engine and nothing more.

Following pattern should be used:

~~~
```{r <label>, dia='<engine>', fig.cap='<drawing name>'}
  to_diagram(src='<src_file_path>')
```
~~~

where:

- `<label>` is a name for a drawing reference, may be omitted
- `<engine>` is a name for a rendering engine, see table for allowed values and corresponding engines
- `<drawing name>` is a caption for drawing, mey be omitted
- `<src_file_path>` is a path to file with diagram data, relative to file into which it's embedded.
  If you are using suggested locations to place docs and diagrams (`docs_src/doc-<subject>` and `docs_src/diagrams`) then value for `<src_file_path>` should be starting like this: `../diagrams/`

4.1.3 Diagrams from Rmd outer file

This is a proposed way to describe diagrams in outer files. Diagram data should be in `.Rmd` file and described as specified in `docs_src/diagrams/README.md` TODO.

This way you'll be able to use Keenwrite to preview diagrams live while editing, which is very handy.

Besides that, you can supply a diagram with additional info, capture decisions that were taken, add TODOs, versions info, etc. along with diagram's data in the same file.

Following pattern should be used:

~~~
```{r <label>, dia='from_src', fig.cap='<drawing name>'}
  to_diagram(src='<src_file_path>')
```
~~~

where:

- `<label>` is a name for a drawing reference, may be omitted
- `<drawing name>` is a caption for drawing, mey be omitted
- `<src_file_path>` is a path to file with diagram data, relative file into which it's embedded.
  If you are using suggested locations to place docs and diagrams (`docs_src/doc-<subject>` and `docs_src/diagrams`) then value for `<src_file_path>` should be starting like this: `../diagrams/`

  Take a NOTE: A `from_src` is specified as engine in this case since an actual engine name is already specified in `.Rmd` file with diagram data.

4.1.4 Get then embed

This is the case if you want to customize diagrams embedding, like aligning, multiple diagrams in a single row, preprocess diagrams before embedding them, etc.

The process is divided into few phases:

1. Download one or multiple rendered images for your diagrams
2. Preprocess diagrams if required

3. Embed them into document

   NOTE: diagrams are still described in a text form

Following pattern should be used:

~~~
# Download diagram
```{r dia='from_src'}
  to_diagram(src='<src_file_path>', downloadOnly=TRUE, downloadName='<file_name>')
~~~

# Download more diagrams if required (like the one above)
...

# Process diagram(s) if required
# Use code chunks for this (R, bash, python and many more with necessary commands)
...

# Embed diagram(s)
```{r <label>, echo=FALSE, fig.cap='<drawing name>'}
  knitr::include_graphics('generated/<file_name>.<file_ext>')
```
```

> NOTE: Example code is shown for outer Rmd file but there is additional `downloadOnly` and `downloadName` arguments are specified in the first code chunk.
> Inline diagram description and raw outer file may be used aswell.

where:

- `<label>` is a name for a drawing reference, may be omitted
- `<drawing name>` is a caption for drawing, mey be omitted
- `<file_name>` is a name that is given to downloaded diagram
- `<file_ext>` is an extension is given depending on output format and optional arguments.
  By default it's:
  - `.svg` for HTML output
  - `.pdf` for PDF output
  - `.png` for other kinds

> Take a NOTE: For postprocessing you can use R, Python, whatever can be called from BASH and many more (depends on your docker image).
```

More on **R chunks**: https://bookdown.org/yihui/rmarkdown/r-code.html, this and this
More on other formats: https://bookdown.org/yihui/rmarkdown/language-engines.html

## 4.2 HTML: SVG - Raw vs from File

By default diagrams are rendered as SVG for HTML output. This format is supported by all rendering engines in Kroki, and produces great results for HTML.

But there is two ways how it's embedded into HTML. It's defined with additional `rawsvg` argument for to_diagram function.

### 4.2.1 Raw SVG

By default SVG content is inserted as a code right into output HTML file and this is what `Raw` is.

Specify optional `rawsvg` argument as `TRUE` for to_diagram function or just ommit it.

PROs:

- Text in SVG becomes searchable and selectable.

CONs:

- Diagrams aren't treated as images anymore and you can't use 'save as' by clicking on them.
  But in this case caption for the diagram will be a reference to an SVG file, so you are still able to save diagrams as files
- You'll have to find your way to post process it if necessary (It's still possible, but I can't offer 'ready to use' recipe yet, and as always there are few ways to do it)
- Lack of customizations (only width, height and alignment available)
  (Probably it could be workarounded but no recipe yet)

Since (IMO) strong customizations and postprocessing is hardly a case for common development docs, RAW svg is used by default.

### 4.2.2 SVG from file

Rendered SVG also can be inserted into HTML as a normal image. It still looks neat when scaled, but now it's just a picture.

Specify optional `rawsvg` argument as `FALSE` for to_diagram function.

This way you'll be able to apply more customizations and do post-processing if necessary.

## 4.3 Customizations

Sometimes you'll want to or need to specify a customizations like width or hight of diagrams when defaults don't work fine or just because of your taste or requirements. Also sometimes you'll have to use non-default image format for the same reasons. Or you may want to go deeper with customizations and define alignment and other things like post processing, multiple images in a row, etc.

> Probably you wouldn't want to use customizations at all when you in the beginning or in hurry to capture your ideas, but if you need some - start from here.

Most of customizations are specified via **R chunk** options.

In case if it's required to download rendered diagram in format other than SVG use `dformat` argument for to_diagram function.

Also for PDF production you'll want for sure things like page break, page orientation specification etc. In this case you will need to add some latex commands into document.

### 4.3.1 R Chunks

**R Chunks** is a way to embed graphics data in **R Markdown**.

> Another way is to use markdown syntax like `![<alt text>](<path to image>){<options>}`, but it's not the case since it works for ready to use images only.

Customizations to the way an image is embded are speicifed via **R chunk** options.

More info about options could be found here and here is a nice cheatsheet.

Following options have effect for diagrams:

- out.width
- out.height
- TODO

Following options have no effect for diagrams:

- fig.width
- fig.height
- TODO

The other way to customize is to use hooks. It's very creative way and all depends on your needs.

Besides spcifying options explicitly for every chunk, you can define default values globally or via named templates, use option hooks to automatically derive one options from another. See more about this in the following sections.

### 4.3.2 Global options

This is done by putting chunk of code with `knitr::opts_chunk$set` function call in the beginning of your documentation (i.e. `index.Rmd`).

Also global options could be redefined later in documentation for the following chunks. For example this could be done in the beginning of every chapter.

> ATTENTION: But make sure in that case that you follow this pattern throughout whole document, otherwise you may find yourself fighting with customizations.

Global options could be good for output format related customizations, in other cases it's better to use templates, described in next section.

Example code to define chunk options globally:

~~~
```{r, include=FALSE}
knitr::opts_chunk$set(
~~~

```
  out.width = '5cm'
)
```
```

More info could be found here

4.3.3 Options templates

This is done by putting chunk of code with `knitr::opts_template$set` function call in the beginning of your documentation (i.e. `index.Rmd`).

Also new templates could be defined later in documentation for the following chunks. For example this could be done in the beginning of every chapter.

Example code to define options template:

````
# Define template
```{r, include=FALSE}
knitr::opts_template$set(fullwidth = list(
  out.width = '100%'
))
```

# Embed diagram, using template for options
```{r, dia='from_src', opts.label='fullwidth'}
to_diagram(src='...')
```

# More options for diagram and/or template's defaults override
```{r, dia='from_src', opts.label='fullwidth', fig.ext='png', out.width='10cm'}
to_diagram(src='...')
```
````

More info could be found here

4.3.4 Options hooks

With options hooks you can define dependencies for chunk options with code in **R**.
````

You can even define hooks for your own options and specify this options for chunks. For example hook for `dia` option is defined in `_fun.R` and some chunk's options depends on it.

This is done by putting chunk of code with `knitr::opts_hooks$set` function call in the beginning of your documentation (i.e. `index.Rmd`).

Also new hooks cold be added or hooks could be redefined later in documentation for the following chunks. For example this could be done in the beginning of every chapter.

Example code to add options hooks:

~~~
```{r, include=FALSE}
# Define hook for existing option out.width
knitr::opts_hooks$set(out.width = function(options){
  # Some sanity checks
  if (!identical(options$out.width, NULL)) {
    # Our function: if out.width is set to 100% then clear out.height option
    if (options$out.width == "100%") {
      options$out.height <- NULL
    }
  }
  # Return new set of options
  options
})

# Define hook for new option fullwidth
knitr::opts_hooks$set(fullwidth = function(options){
  # Some sanity checks
  if (!identical(options$fullwidth, NULL)) {
    # Our function: if fullwidth is defined and evaluated as TRUE then set out.width t
    if (fullwidth) {
      options$out.width <- "100%"
    }
  }
  # Return new set of options
  options
})
```
~~~

~~~
# Embed diagram, using new option to define width
```{r, dia='from_src', fullwidth=TRUE}
to_diagram(src='...')
```
~~~

4.3.5 Multiple output formats

If you are intended to produce documentation in multiple output formats, most probably you'll have to apply different customizations for different formats.

This could be easily achived by using **R** `if ... else ...` pattern (works like ternary operator) when applying values to options, or with a few lines **R** code in complicated cases.

For example:

~~~
```{r, dia='from_src', out.height=if (!knitr::is_html_output()) '15cm' else NULL}
to_diagram(src='...')
```
~~~

will set drawing's height to 15 centimeters for output formats other than HTML.

4.3.6 Define parameters

There is a way to separate actual values for options from code via parameters, that are defined in a front-matter part of `index.Rmd`.

This way you could tune your output or other aspects of your documentation just by adjusting those values and keep them all in a single place.

> By the way - there is already some params defined that are reqruired for to_diagram function and related hooks.

Values for parameters can be calculated just in time with **R** by defining their values using following pattern: `` `r <R expressions>` ``, replacing `<R expressions>` with your **R** code.

Parameters definition example:

```
# This is front-matter's section to define params
params:
  # Static value:
  background: "#FFFFFF"
  # Calculated value:
  current_date: `r Sys.Date()`
  # Output format depended value:
  fig_ext: `r if (knitr::is_latex_output()) 'png' else null`
```

Parameters usage example:

~~~
```{r, dia='from_src', fig.ext=params$fig_ext}
to_diagram(src='...')
```
~~~

More info can be found here.

4.3.7 Putting all together

To avoid redundant work, and to ease refactoring and tuning following methodology is suggested to use from the start to benefit from **R Markdown** features:

1. Declare subject specific parameters for your documentation
2. Define global options if necessary, use parameters to adjust options
3. Define all the necessary options templates for your diagrams embedding. You could use differnet templates for different output formats and specify their names via params, or you could define universal templates and tune them via params or use `if (knitr::is_<html/latex>_otput()) <value1> else <value2>` pattern.
4. Use options templates for diagrams code chunks, providing extra options if required

4.4 Data format

By default following operations are made when diagram drawing is embedded into documentation:

1. Diagram drawing is downoaded in **svg** format
2. If output format is other than HTML then data is converted locally with **rsvg** tool into one of the following:

- **pdf** for PDF output
- **png** for the rest

In some circumstances conversion with **rsvg** may have artifacts, and in this case data format can fix the problem.

Usualy it’s better to set format for downoaded drawing, but not all renderers may support it, so you may try to change local side conversion output format.

> NOTE: some rendering engines like **mermaid** or **pikchr** produce only **svg** and **rsvg** can’t handle it right

4.4.1 Renderer format

Speicfy format for downloaded drawing. Even though it should produce better result than the case below, not all rendering engines may support required format. See diagram types

```
to_diagram(... , dformat='<format>')
```

Possible values for `<format>`:

- `png` - loseless image, can be used for most output documentation formats
- `jpeg` - lossy image, can be used for most output documentation formats
- `svg` - for HTML output only, but can be locally converted to **png**, **jpeg** or **pdf** after downloading
- `pdf` - for PDF output only

4.4.2 Local conversion

This kind of conversion is possible only in case if diagram drawings are rendered and downloaded in **svg** format.

~~~
```{r <label>, dia='from_src', fig.ext='<format>'}
  to_diagram(...)
```
~~~

Possible values for `<format>`:

- `png` - loseless image, can be used for most output documentation formats
- `jpeg` - lossy image, can be used for most output documentation formats
- `pdf` - for PDF output only

4.5 PDF production

PDF document production most probably will require more attention.

Make sure you are satisfied with your latex class for PDF output, to avoid redundant work.

Then you'll want for sure to limit hight for large drawings, but don't hurry to do this.

It's better to start with splitting the pages and setting necessary layout first.

4.5.1 Page breaks

Every chapter is started from new page.

For additional page breaks just add the following line into document:

```
\newpage
```

4.5.2 Page orientation

Make sure following lines are included into your latex preamble part:

```
\usepackage{lscape}
\usepackage{pdfpages}
```

Then in documentation use following pattern:

```
```

Some text on a portrait page.
```
```

~~~
\newpage
\blandscape

Some text on a landscape page.

\newpage
\elandscape

Some other text on a portrait page.
```
~~~

4.6 Full to_diagram function specification

Function **to_diagram** is defined in `docs_src/_fun.R` file which should be included into `index.Rmd` file. More about this is TODO: devdoc-swissknife anatomy.

This function renders diagram from textual description and inserts it into document. Diagram is inserted with `knitr::include_graphics` function or as raw SVG code for HTML output (raw svg can be turned off.

Here is it’s definition:

```
to_diagram  <- function(
    data = "", src = "",
    dformat = "", rawsvg = TRUE,
    downloadOnly = FALSE, downloadName = "",
    service="Kroki", serviceUrl=NULL,
    engine=NULL,
    page=NULL,
    force=FALSE
    )
```

Arguments:

```
data       - Diagram description in text form (should be compatible with
                  rendering engine).
                  Ignored if value for `src` is specified.

src         - Path to outer file with diagram description in text form.
```

```
                See `docs_src/diagrams/README.md` for details.
                Should be omitted if `data` argument is to be used.

dformat       - Specify data format for downloaded diagram.
                See https://kroki.io/#support for details.
                `svg` would be used by default if value is empty string.
                Sometimes SVG conversion to PDF/PNG not works good on client
                side (default). In that case it's suggested to use PNG or PDF.
                Take a NOTE: not all Kroki services provide diagrams
                in PNG / PDF format.

rawsvg        - Option to insert SVG as code right into HTML.
                This way text on diagrams is searchable.
                Enabled (TRUE) by default.
                Set to FALSE to embed SVG as image.
                Applicable only for HTML output and only if `dformat`
                is "SVG" or empty string.

downloadName  - File name (part before extension) for local diagram's files.

downloadOnly  - Only generate and download diagram, don't insert into doc.
                Use for custom diagram embedding later in the doc.
                All the necessary format conversions will be completed.
                Result would be in `docs_src/generated` dir.
                If `downloadName` is specified then it would be used for
                file name, otherwise label would be used if diagram is
                specified via data, otherwise file name would be generated
                from `src` path.
                File extension would depend on `dformat` argument and on
                `fig.ext` options for R chunk

service       - Rendering service (now it's sonly "Kroki")

serviceUrl    - Rendering service base url.
                This is starting part of url for service access.
                Default is path `http://kroki:8000` to local Kroki
                instance that is started with `make_doc.sh`

engine        - Diagram rendering engine. Specify one of supported engines
                (see Kroki for engines list)
                Special value `from_src` for outer files SHOULD be used
                to get engine from source file itself in case
                if recommendations from `docs_src/diagrams/README.md`
                are satisfied.
```

```
           If omitted or NULL then value of R chunk option named `dia`
                is treated as `engine`

page      - Page for rendering. For use with multipage office documents

force      - If FALSE(default) then existing downloaded diagram data
          would be used. Set to TRUE to regenerate diagram from scratch
```

Chapter 5

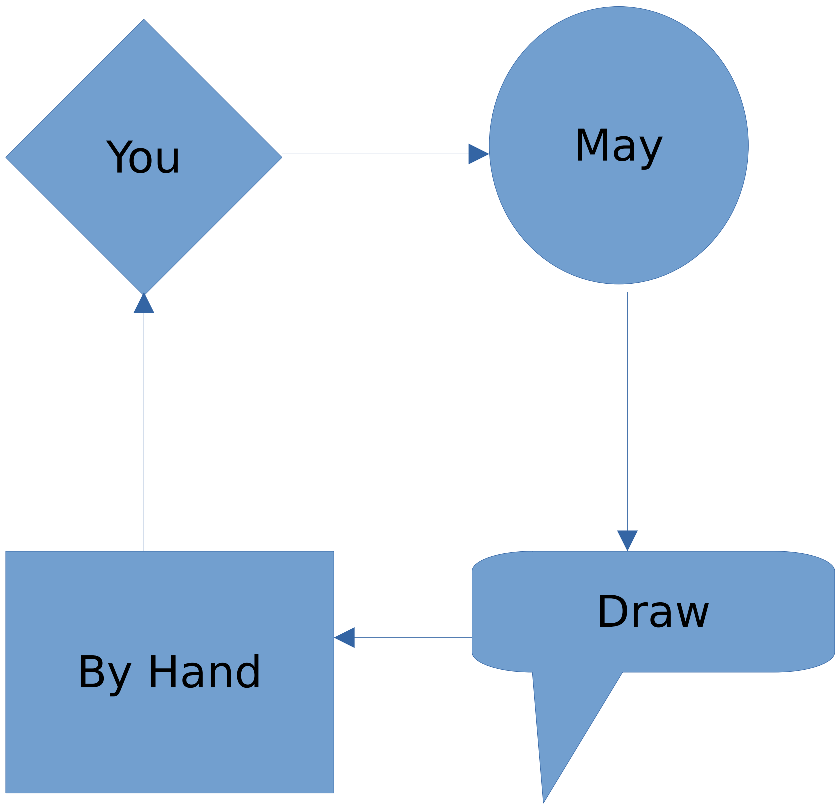
Other drawings

5.1 Already rendered diagrams

5.2 Office files

Office files could be used as source for diagrams

~~~
```{r, examples-libre-draw-singlepage, dia="draw"}
  to_diagram(src = "../diagrams/examples/libre-draw-singlepage.odg", service = "Office
```
~~~



This is intended for drawings in first order, but actualy this could be any document that can be opened by Libre Office.

5.3 Add your custom renderer

Chapter 6

Project anatomy

Chapter 7

Diagrams reference

7.1 Refrencing embeded diagrams

https://www.rdocumentation.org/packages/knitr/versions/1.33/topics/fig_chunk

7.2 Native R Markdown

https://bookdown.org/yihui/rmarkdown-cookbook/diagrams.html