The Notebookinator

Easy formatting for robotics notebooks.

v0.1.0

https://github.com/BattleCh1cken/notebookinator

Abstract

Welcome to the Notebookinator, a Typst package meant to simply the notebooking process for the Vex Robotics Competition. Its theming capabilities handle all of the styling for you, letting you jump right into writing documentation.

While it was designed with VRC in mind, it could just as easily be used for other competitor systems such as the First Robotics Competition and the First Tech Challenge.

The Notebookinator

Contents

1. Installation

The best way to install the Notebookinator is as a local package. Make sure you have the following software installed on your computer:

- Typst
- Git
- VSCode

Once you have those, find the matching command for you operating system, and then run it:

Linux:

```
git glone https://github.com/BattleCh1cken/notebookinator \
~/.local/share/typst/packages/local/
```

MacOS:

```
git glone https://github.com/BattleCh1cken/notebookinator \
~/Library/Application Support/typst/packages/local/
```

Windows:

```
git glone https://github.com/BattleCh1cken/notebookinator %APPDATA%
\typst\packages\local\
```

2. Basic Usage

Once the template is installed, you can import it into your project like this:

```
#import "@local/notebookinator": *
```

Once you've done that you can begin to write your notebook:

```
#import themes.default: default_theme, components

#show: notebook.with(theme: default_theme)

#create_frontmatter_entry(title: "Table of Contents")[
  #components.toc()
]
```

3. API Reference

3.1. Template

3.1.1. notebook

The base notebook template. This function is meant to be applied to your entire document as a show rule.

Example Usage:

```
#import themes.default: default_theme

#show: notebook.with(
  theme: default_theme
)
```

3.1.1.1. Parameters

```
notebook(
  team_name: string,
  season: string,
  year: string,
  cover: content,
  theme: theme,
  body: content
) -> content
```

team_name `string`

The name of your team.

Default: `none`

season `string`

The name of the current season

Default: `none`

year `string`

The years in which the notebook is being written

Default: `none`

cover `content`

the title page of the notebook

Default: `none`

theme `theme`

The theme that will be applied to all of the entries. If no theme is specified it will fall back on the default theme.

Default: `(:)`

body `content`

The content of the notebook. This will be ignored. Use the create_entry functions instead.

3.2. Entries

3.2.1. create_entry

Generic entry creation function.

3.2.1.1. Parameters

```
create_entry(
  entry_type: string,
  title: string,
  type: string,
  start_date: datetime,
  end_date: datetime,
  body: content
)
```

entry_type `string`

The type of entry. Takes either “frontmatter”, “body”, or “appendix”.

Default: `none`

title `string`

The title of the entry.

Default: `""`

type `string`

The type of entry. The possible values for this are decided by the theme.

Default: `none`

start_date `datetime`

The date that the entry started at.

Default: `none`

end_date `datetime`

The date that the entry ended at. If not specified, it will fall back on the `start_date`.

Default: `none`

body `content`

The content of the entry.

3.2.2. create_frontmatter_entry
Variant of the `#create_entry()` function that creates a frontmatter entry.

3.2.2.1. Parameters
```
create_frontmatter_entry()
```

3.2.3. create_body_entry
Variant of the `#create_entry()` function that creates a body entry.

3.2.3.1. Parameters
```
create_body_entry()
```

3.2.4. create_appendix_entry
Variant of the `#create_entry()` function that creates an appendix entry.

3.2.4.1. Parameters
```
create_appendix_entry()
```

3.3. Glossary

3.3.1. add_term
Add a term to the glossary

3.3.1.1. Parameters
```
add_term(
  word: string,
  definition: string
)
```

word `string`

The word you're defining

definition `string`

The definition of the word

3.4. Additional Datatypes

3.4.1. Themes

3.4.2. Entries

3.5. Radial Theme
The Radial theme is a minimal theme focusing on nice, rounded corners.

3.5.1. toc
Print out the table of contents

-> content

3.5.1.1. Parameters

```
toc()
```

3.5.2. glossary

Print out the glossary -> content

3.5.2.1. Parameters

```
glossary()
```

3.5.3. admonition

A message in a colored box meant to draw the reader’s attention.

3.5.3.1. Parameters

```
admonition(
  type: string,
  body: content
) -> content
```

type `string`

The type of admonition. Available types include:
- “note”
- “example”
- “quote”
- “equation”
- “decision”
- “build”

Default: `none`

body `content`

The content of the admonition

3.5.4. pro_con

A table displaying pros and cons.

3.5.4.1. Parameters

```
pro_con(
  pros: content,
  cons: content
) -> content
```

pros `content`

The positive aspects

Default: `[]`

cons `content`

The negative aspects

Default: `[]`

3.5.5. decision_matrix

A decision matrix table.

3.5.5.1. Parameters

```
decision_matrix(
  properties: array,
  ..choices: array
) -> content
```

properties `array`

A list of the properties that each choice will be rated by

Default: `none`

..choices `array`

An array containing the name of the choices as its first member, and values for each of the properties at its following indices

3.5.6. tournament

A Series of tables displaying match data from a tournament. Useful for tournament analysis entries.

3.5.6.1. Parameters

```
tournament(..matches: dictionary) -> content
```

..matches `dictionary`

A list of all of the matches at the tournament. Each dictionary must contain the following fields:

- match (string) The name of the match
- red_alliance (dictionary) The red alliance
  - teams (array)
  - score (integer)
- blue_alliance (dictionary) The blue alliance
  - teams (array)
  - score (integer)
- won (boolean) Whether you won the match
- auton (boolean) Whether you got the autonomous bonus
- awp (boolean) Whether you scored the autonomous win point
- notes (content) Any additional notes you have about the match

4. Developer Documentation

4.1. Project Architecture

The Notebookinator is split into two sections, the base template, and the themes. The base template functions as the backend of the project. It handles all of the information processing, keeps track of global state, makes sure page numbers are correct, and so on. It exposes the main API that the user interacts for creating entries and creating glossary entries.

The themes act as the frontend to the project, and are what the user actually sees. The themes expose an API for components that need to be called directly inside of entries. This could include things like admonitions, charts, and decision matrices.

4.2. File Structure

- `lib.typ`: The entrypoint for the whole template.
- `internals.typ`: All of the internal function calls that should not be used by theme authors or users.
- `entries.typ`: Contains the user facing API for entries, as well as the internal template functions for printing out the entries and cover.
- `glossary.typ`: Contains the user facing API for the glossary.
- `globals.typ`: Contains all of the global variables for the entire project.
- `utils.typ`: Utility functions intended to help implement themes.
- `themes/`: The folder containing all of the themes.
  - `themes.typ`: An index of all the themes are contained in the template
- `docs.typ`: The entry point for the project documentation.
- `docs-template.typ`: The template for the project documentation.

4.3. Implementing Your Own Theme

The following section covers how to add a theme to the ones already in the template. It only covers how to write the code, and not how to get it merged into the main project. If you want to learn more about our contributing guidelines, check our `CONTRIBUTING.MD` file in our GitHub.

4.3.1. Creating the Entry Point

This section of the document covers how to add your own theme to the template. The first thing you'll have to do is create the entry point for your theme. Create a new directory inside the `themes/` directory, then create a Typst source file inside of that directory. For example, if you had a theme called `foo`, the path to your entry point would look like this: `themes/foo/foo.typ`.

Once you do this, you'll have to add your theme to the `themes/themes.typ` file like this:

```
#import `./foo/foo.typ`
```

Do not use a glob import, we don't want to pollute the namespace with the functions in the theme.

4.3.2. Implementing Theme Functions

Next you'll have to implement the functions contained inside the theme. These functions are all called internally by the template. While we recommend that you create implementations for all of them, if you omit one the template will fall back on the default theme.

The first functions you should implement are the ones that render the entries. You'll need three of these, one for each type of entry (frontmatter, body, and appendix).

Each of these functions must take a context parameter, and a body parameter. The context parameter provides important information, like the type of entry, and the date it was written at. The body parameter contains the content written by the user.

The template expects that each of these functions returns a `#page()` as content.

Here's a minimal example of what these functions might look like:

```
#let frontmatter_entry(context: (:), body) = {
  show: page.with(
    header: [ = Frontmatter header ],
    footer: counter(page).display("i")
  )

  body
}
```

```
#let body_entry(context: (:), body) = {
  show: page.with(
    header: [ = Body header ],
    footer: counter(page).display("1")
  )

  body
}
```

```
#let appendix_entry(context: (:), body) = {
  show: page.with(
    header: [ = Appendix header ],
    footer: counter(page).display("i")
  )

  body
}
```

Next you'll have to define the rules. This function defines all of the global configuration and styling for your entire theme. This function must take a doc parameter, and then return that parameter. The entire document will be passed into this function, and then returned. Here's and example of what this could look like:

```
#let rules(doc) = {
  set text(fill: red) // Make all of the text red

  doc // Return the entire document
}
```

Then you'll have to implement a cover. The only required parameter here is a context variable, which stores information like team number, game season and year.

Here's an example cover:

```
#let cover(context: (:)) = [
  #set align(center)
  *Foo Cover*
]
```

4.3.3. Defining the Theme

Once you define all of your functions you'll have to actually define your theme. The theme is just a dictionary which stores all of the functions that you just defined.

The theme should be defined in your theme's entry point (`themes/foo/foo.typ` for this example).

Here's what the theme would look like in this scenario:

```
#let foo_theme = (
  // Global show and set rules
  rules: rules,

  cover: cover,

  // Entry pages
  frontmatter_entry: frontmatter_entry,
  body_entry: body_entry,
  appendix_entry: appendix_entry
)
```

4.3.4. Creating Components

With your base theme done, you may want to create some additional components for you to use while writing your entries. This could be anything, including graphs, tables, Gantt charts, or anything else your heart desires.

4.4. Utility Functions

4.4.1. print_toc

Utility function to help themes implement a table of contents

4.4.1.1. Parameters

```
print_toc(
  type: string,
  callback: function
) -> content
```

type `string`

Takes either "frontmatter", "body", or "appendix"

Default: `"body"`

callback `function`

A function which takes the context of the entry as input, and returns the content for a single row

4.4.2. print_glossary

A utility function meant to help themes implement a glossary

4.4.2.1. Parameters

print_glossary(callback: `function`) -> `content`

callback `function`

A function returning the content of a single glossary entry

4.4.3. calc_decision_matrix

A utility function that does the calculation for decision matrices for you

Example Usage

```
calc_decision_matrix(
  properties: ("Versatility", "Flavor", "Chrunchiness"),
  ("Sweet potato", 2, 5, 1),
  ("Red potato", 2, 1, 3),
  ("Yellow potato", 2, 2, 3),
)
```

The function returns an array of dictionaries, one for each choice. Each dictionary contains the name of the choice, the values for each property, the total, and whether the choice has the highest score or not. Here's an example of what one of these dictionaries might look like:

```
  (
    name: "Sweet potato",
    values: (2, 5, 1),
    total: 8,
    highest: true,
  )
```

4.4.3.1. Parameters

```
calc_decision_matrix(
  properties: array string,
  ..choices: array
) -> array
```

properties `array string`

A list of the properties that each choice will be rated by

Default: `()`

..choices `array`

All of the choices that are being rated. The first element of the array should be the name of the

4.4.4. change_icon_color

Returns the raw image data, not image content You’ll still need to run image.decode on the result

4.4.4.1. Parameters

```
change_icon_color(
  raw_icon: string,
  fill: color
) -> string
```

raw_icon `string`

The raw data for the image. Must be svg data.

Default: `""`

fill `color`

The new icon color

Default: `red`

4.4.5. colored_icon

Takes the path to an icon as input, recolors that icon, and then returns the decoded image as output.

4.4.5.1. Parameters

```
colored_icon(
  path: string,
  fill: color,
  width: ratio length,
  height: ratio length,
  fit: string
) -> content
```

path `string`

The path to the icon. Must point to a svg.

fill `color`

The new icon color.

Default: `red`

width `ratio length`

Width of the image

Default: `100%`

height `ratio` `length`

height of the image

Default: `100%`

fit `string`

How the image should adjust itself to a given area. Takes either “cover”, “contain”, or “stretch”

Default: `"contain"`