innoConv Documentation

Release 0.0.4

Innocampus

Apr 15, 2019

CONTENTS

CHAPTER

ONE

TABLE OF CONTENTS

1.1 What is innoConv?

innoConv is a converter for educational content.

The software transforms source content into an intermediate JSON representation that can be displayed with the help of an *innodoc-compatible viewer*.

It takes plain-text files as a source. These are written in the Markdown language and stored in a particular *directory structure* reflecting the sections and subsections of the work.

See also:

Check out the *example course* to see how it looks like.

1.1.1 Features

Common features as basic text formatting, links, tables, lists, etc. are already provided by Markdown out-of-the-box.

While staying as close to traditional Markdown as possible innoConv supports a variety of additional constructs. Many of them are targeted specifically at the creation of educational content.

These include

- *Localization*
- Math formulas
- Images and videos
- *Interactive exercises*
- *PGF/TikZ*
- Table of contents
- *Inter-section references*
- *Glossary*

1.1.2 innoDoc

innoConv is a part in the software package innoDoc. It handles the translation of source content to the an intermediate JSON represenation.

innoConv does neither have any busisness with how content is displayed nor helps in its creation. Instead it leaves these tasks completely to others in the processing chain.

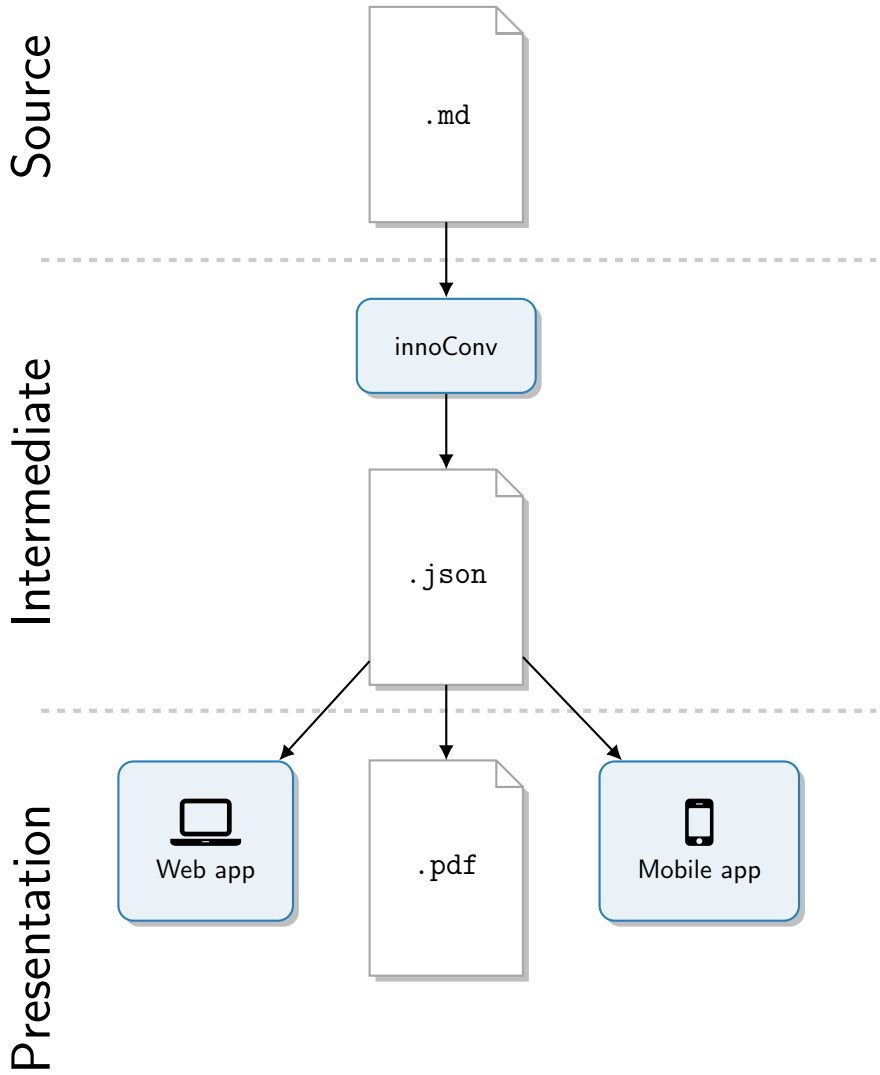

Fig. 1: Overview of the innoDoc software architecture.

See also:

See section *Content creation* for a in-depth discussion on how to write course content.

1.1.2.1 Viewers

At the moment there are two viewers in development.

innodoc-webapp React-based HTML5 web application

innodoc-app React Native-based Smartphone App

Note: Configuration and deployment of viewers is not the scope of this document. Please refer to the respective documentation.

1.2 Getting started

1.2.1 Prerequisites

innoConv is mainly used on Linux machines. It might work on Mac OS and Windows/Cygwin/WSL. You are invited to share your experiences.

1.2.2 Dependencies

The only dependencies you have to provide yourself is Pandoc and the Python interpreter.

1.2.2.1 Python 3

innoConv is being tested and developed with **Python 3.4-3.7**.

Python should be available on the majority of Linux machines nowadays. Usually it's being installed using a package manager.

1.2.2.2 Pandoc

You need to make sure to have a recent version of the **`pandoc`** binary available in `$PATH` (version 2.7 at the time of writing). There are several ways how to install Pandoc.

1.2.3 Installation

1.2.3.1 Using pip

The easiest way to install innoConv is to use **`pip`**.

Given you have a regular Python setup with **`pip`** available the following installs innoConv in your user directory (usually `~/.local` under Linux).

```
$ pip install --user innoconv
```

For the **innoconv** command to work, make sure you have `~/.local/bin` in your `$PATH`.

For a system-wide installation you can omit the `--user` argument.

```
$ pip install innoconv
```

1.2.3.2 In a virtual environment

It's possible to install innoConv into a virtual environment. Setup and activate a virtual environment in a location of your choice.

```
$ python3 -m venv /path/to/venv
$ source /path/to/venv/bin/activate
```

Install innoConv in your virtual environment using **pip**.

```
$ pip install innoconv
```

If everything went fine you should now have access to the **innoconv** command.

The next time you login to your shell make sure to activate your virtual environment before using **innoconv**.

1.3 How to use innoConv

The principle way of using innoConv is the CLI (Command-line interface) **innoconv**. Another option is to use innoConv in a *programmatic way* as Python library.

Run the converter on your content directory.

```
$ innoconv /path/to/my/content
```

This will trigger a conversion and store the result in a folder `innoconv_output`. A return code other than `0` indicates an unsuccessful run.

Note: According to Unix convention you will not see any messages if the conversion was successful. Though you might pass the *`--verbose`* flag to change this behavior.

1.3.1 Command line arguments

1.3.1.1 innoconv

Converter for interactive educational content.

```
innoconv [OPTIONS] SOURCE_DIR
```

Options

-o, --output-dir <output_dir>
: Set output directory. [default: ./innoconv_output]

-e, --extensions <extensions>
Enable extensions (comma-separated). [default: copy_static,generate_toc,join_strings,tikz2svg,write_manifest]

-f, --force
Force overwriting of output.

-v, --verbose
Print verbose messages.

--version
Show the version and exit.

Arguments

SOURCE_DIR
Required argument

1.3.2 Using innoConv as a library

Using innoConv within a Python program involves creating a *Manifest* object and using it with a *InnoconvRunner*.

```
>>> manifest = Manifest(manifest_data)
>>> runner = InnoconvRunner(source_dir, output_dir, manifest, extensions)
>>> runner.run()
```

Have a look at the source of *innoconv.cli* for a more detailed example.

1.4 Content creation

This section deals with the creation of content files that are fed into innoConv for processing.

As already mentioned, the main format for writing content is Markdown. The available references for the Markdown language generally do apply for innoConv but we will direct you specifically to Pandoc's Markdown. The reason for that is Markdown exists in different flavours and innoConv is using Pandoc under the hood.

See also:

The *Example course* serves as a reference to authors.

1.4.1 Best practices

1.4.1.1 Text editor

Content is written in plain-text. Therefore you will need a text editor to author innoDoc content. The choices are endless. If in doubt your operating system comes with a text editor pre-installed.

Warning: Please use *UTF-8 character encoding* exclusively when writing documents for innoConv. Make sure your editor uses the right encoding.

1.4.1.2 Version control

Writing large amounts of text is often a joint effort with a lot of edits and many contributors. Using a VCS (Version control system) (e.g. git) for personal use is highly recommended. On a collaborative project it's indispensable.

1.4.2 Writing content

Your content typically resides in a dedicated directory referred to as *root directory* from now on. There are some conventions that you need to follow. These are explained in this section.

1.4.2.1 The manifest file

The root directory is home to the `manifest.yml` file. It is used to store meta information about your content, like the title, the languages and so on.

Listing 1: A minimal example for a content manifest.

```
title:
  en: Example course
  de: Beispielkurs
languages: en,de
```

Note: If your content uses only one language, you will still need to put this single language here.

1.4.2.2 Directory and file structure

Sections and subsections

In the previous section we saw how to specify the available languages for the content. For every language one subdirectory needs to exist in the root directory.

Under each language directory there is a structure of folders reflecting the part/chapter/section structure of the text.

Note: Every directory needs one `content.md`.

The names of the directories determine the order in the actual text. They are sorted alphanumerically. The directory name itself can be used to create *cross-references* from one part in the the text to another. Also, they might appear in more technical contexts such as URLs (Uniform Resource Locator).

Note: While technically not strictly required, for convenience it's advisable to limit directory names to characters, numbers, hyphen and underscore (`a-z`, `0-9`, `-` and `_`).

Listing 2: Example directory structure for two languages.

```
root
├── manifest.yml
├── en
|   ├── content.md
```

(continues on next page)

(continued from previous page)

```
|   ├── 01-part
|   |   ├── content.md
|   |   ├── 01-section
|   |   |   └── content.md
|   |   └── ...
|   └── ...
└── de
    ├── content.md
    ├── 01-part
    |   ├── content.md
    |   ├── 01-section
    |   |   └── content.md
    |   └── ...
    └── ...
```

Important: The directory structure in each of the language folders need to match!

Static files

There can be optional directories `_static` for media files.

These can exist in two different locations: Either at the root folder or inside a language folder. Some files might have a translated version. To account for this a localized version of the file can be put in the language's static folder.

Listing 3: Locations for static files.

```
root
├── _static
|   ├── chart.svg
|   └── image.png
├── en
|   ├── _static
|   |   └── video.mp4
└── de
    └── _static
        └── video.mp4
```

For the sake of clarity other needed files and directories are omitted in this listing.

1.4.2.3 Content files

A file `content.md` needs to exist in every section folder. It has a small section at the top of the file called YAML metadata block that contains the section title.

Listing 4: Example YAML metadata block.

```
---
title: Example title for this section
---
```

After the metablock you can write your actual content.

Note: A `content.md` needs to exist for every language version, e.g. `en/section01/content.md` and `de/section01/content.md`.

This section will not provide an exhaustive list of formatting options. Instead it will mainly focus on some features that are unique to innoDoc.

See also:

All possibilities are documented in the *example course*.

Media files

Todo: Media files

PGF/Ti*k*Z

PGF/Ti*k*Z is used to create vector graphics and is written in TeX.

Todo: pgf/tikz example

Interactive exercises

Todo: section interactive exercises

Cross-references

Todo: section cross-references

Glossary

Todo: section glossary

1.4.3 Localization

Todo:

- general words

- use with only one language

See also:

- Section *Sections and subsections* on how to structure directories with multiple languages.
- Section *Static files* for translating media files.
- Section *Content files* for translating Markdown content.

1.4.4 Example course

There's an example course. It's a comprehensive demonstration of what is possible with innoConv.

It serves the following purposes:

- Showcase the capabilities and features
- Reference for authors
- Material for automatic software tests

Note: If you want to start compiling content, check out this course and start using innoConv right away.

1.4.4.1 Links

- Live version
- Content source repository

1.5 Module overview

1.5.1 innoconv.cli

Command line interface for the innoConv document converter.

class innoconv.cli.**CustomEpilogCommand**(*name*, *context_settings=None*, *callback=None*, *params=None*, *help=None*, *epilog=None*, *short_help=None*, *options_metavar='[OPTIONS]'*, *add_help_option=True*, *hidden=False*, *deprecated=False*)

Format epilog in a custom way.

format_epilog(*_*, *formatter*)

Format epilog while preserving newlines.

1.5.2 innoconv.constants

Project constants.

innoconv.constants.**DEFAULT_OUTPUT_DIR_BASE**

Default innoconv output directory

innoconv.constants.**DEFAULT_EXTENSIONS**
Default enabled extensions

innoconv.constants.**ENCODING**
Encoding used in this project

innoconv.constants.**CONTENT_BASENAME**
Basename for the content file in a section

innoconv.constants.**MANIFEST_BASENAME**
Manifest filename

innoconv.constants.**STATIC_FOLDER**
Static folder name

1.5.3 innoconv.ext

innoConv extensions.

Extensions are a way of separating concerns of the conversion process into independent modules. They can be enabled on a one-by-one basis as not all features are needed in all cases.

Extensions interface with *InnoconvRunner* through a set of methods defined in *AbstractExtension*.

innoconv.ext.**EXTENSIONS = {'copy_static': <class 'innoconv.ext.copy_static.CopyStatic'>,**
List of available extensions

1.5.4 innoconv.ext.abstract

Base class for all other extensions.

The AbstractExtension is not instantiated directly but serves as super-class to all extensions.

class innoconv.ext.abstract.**AbstractExtension**(*manifest*)
Abstract class for extensions.

The class all extensions inherit from. The to-be-implemented methods document the available events that are triggered during the conversion process.

Extension classes should have a `_helptext` attribute. It's used to display a brief summary.

classmethod helptext()
Return a brief summary of what the extension is doing.

extension_list(*extensions*)
Receive list of active extension instances.

Parameters **extensions** (*list*) – List of all active extension instances

start(*output_dir*, *source_dir*)
Conversion is about to start.

Parameters

- **output_dir** (*str*) – Base output directory
- **source_dir** (*str*) – Content source directory

pre_conversion(*language*)
Conversion of a single language folder is about to start.

Parameters **language** (*str*) – Language that is currently being converted.

pre_process_file(*path*)
Conversion of a single file is about to start.

Parameters **path** (*str*) – Output path

post_process_file(*ast*, *title*)
Conversion of a single file finished. The AST can be modified.

Parameters

- **ast** (*List of content nodes*) – File content as parsed by pandoc.
- **title** (*str*) – Section title (localized)

post_conversion(*language*)
Conversion of a single language folder finished.

Parameters **language** (*str*) – Language that is currently being converted.

finish()
Conversion finished.

1.5.5 innoconv.ext.copy_static

Extension that copies static files.

Content can include figures, images and videos. Static files can be included in a special folder named `_static`. The files will be copied to the output directory automatically.

1.5.5.1 Translation

It's possible to have language-specific versions of a static file.

For that to work you need to have a `_static` folder beneath the language folder. Files in this folder will take precedence over the common `_static` folder for that language.

Example: `en/_static/example.png` takes precedence over `_static/example.png` in the English version.

1.5.5.2 Relative and absolute reference

Files can be referenced using relative or absolute paths.

Absolute paths are resolved to the root folder, either the common (`_static`) or language-specific (`en/_static`) folder.

Relative paths are resolved to the root folder but have the chapters path fragment appended.

Example

This example shows how a reference to an image is resolved. The references happen inside the section `chapter01` in the English language version.

Type	Resolution
Relative	`subdir/my_picture.png` → `en/_static/chapter01/subdir/my_picture.png`
Absolute	`/subdir/my_picture.png` → `en/_static/subdir/my_picture.png`

class innoconv.ext.copy_static.**CopyStatic**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Copy static files to the output folder.

This extension copies checks the AST for references to static files and copies them from the content source directory to the output directory.

start(*output_dir*, *source_dir*)
Remember directories.

pre_conversion(*language*)
Remember current conversion language.

pre_process_file(*path*)
Remember file path.

post_process_file(*ast*, _)
Generate list of files to copy.

finish()
Copy static files to the output folder.

1.5.6 innoconv.ext.generate_toc

Extension that generates a table of contents.

A table of contents is generated from the course sections and added to the *Manifest*.

class innoconv.ext.generate_toc.**GenerateToc**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Generate a TOC from content sections.

start(*output_dir*, _)
Remember output directory.

pre_conversion(*language*)
Remember current conversion language.

pre_process_file(*path*)
Remember current path.

post_process_file(_, *title*)
Add this section file to the TOC.

1.5.7 innoconv.ext.join_strings

Merge consecutive sequences of strings and spaces into a single string element.

The motivation behind this extension is to make the AST more readable and also to save space by compressing the representation. The actual appearance in a viewer remains identical.

This extension modifies the AST.

1.5.7.1 Example

```
{"t":"Str","c":"Foo"},{"t":"Space"},{"t":"Str","c":"bar"}]    →    {"t":"Str",
"c":"Foo bar"}]
```

class innoconv.ext.join_strings.**JoinStrings**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Merge consecutive strings and spaces in the AST.

post_process_file(*ast*, _)
Process AST in-place.

1.5.8 innoconv.ext.tikz2svg

Convert and insert Ti*k*Z figures.

SVG files are be rendered from Ti*k*Z code and saved in the folder `_tikz` in the static folder of the output directory.

Ti*k*Z code blocks are replaced by image elements.

Note: In order to use this extension you need to have the following installed on your system:

- LaTeX distribution with PGF/Ti*k*Z
- pdf2svg

1.5.8.1 Example

A Ti*k*Z image is written using a code block.

```
```tikz
\\begin{tikzpicture}
\\shade[left color=blue,right color=red,rounded corners=8pt] (-0.5,-0.5)
  rectangle (2.5,3.45);
\\draw[white,thick,dashed,rounded corners=8pt] (0,0) -- (0,2) -- (1,3.25)
  -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
\\node[white] at (1,-0.25) {\\footnotesize House of Santa Claus};
\\end{tikzpicture}
```
```

Upon conversion, this code block will be replaced in the output with an image tag, similar to the following.

```
![](/_tikz/tikz_abcdef0123456789.svg "Alt text")
```

class innoconv.ext.tikz2svg.**Tikz2Svg**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Convert and insert Ti*k*Z images.

start(*output_dir*, *source_dir*)
Initialize the list of images to be converted.

post_process_file(*ast*, _)
Find TikZ images in AST and replace with image tags.

finish()
Render images and copy SVG files to the static folder.

1.5.9 innoconv.ext.write_manifest

Extension that writes a `manifest.json` to the output folder.

Every course needs a *Manifest*. Additionally to the fields from the source manifest it can include a table of contents and a glossary.

class innoconv.ext.write_manifest.**WriteManifest**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Write a manifest file when conversion is done.

start(*output_dir*, _)
Remember output directory.

finish()
Output course manifest.

1.5.10 innoconv.manifest

The manifest comprises course metadata.

A `manifest.yml` file needs to exist in every course content and resides at the content root directory.

There is also a representation in JSON format. It is generated automatically by the extension *WriteManifest* and copied to the output folder.

Other extensions may add custom fields to the output manifest by implementing a method `manifest_fields()`. It needs to return a `dict` that is merged into the manifest.

A `manifest.yml` is written by course authors while the `manifest.json` is generated by the converter and used by innoconv-compatible viewers. The included information for both versions differ.

1.5.10.1 Example

```
title:
  en: Example title
  de: Beispiel-Titel
languages: en,de
```

class innoconv.manifest.**Manifest**(*data*)
Represents course metadata.

classmethod from_directory(*dirpath*)
Read manifest from content directory.

Parameters **dirpath** (*str*) – Full path to content directory.

Return type *Manifest*

Returns Manifest object

classmethod from_yaml(*yaml_data*)
Create a manifest from YAML data.

Parameters **yaml_data** (*str*) – YAML representation of a manifest

Return type *Manifest*

Returns Manifest object

1.5.11 innoconv.runner

The innoConv runner is the core of the conversion process.

It traverses the source directory recursively and finds all content files. These are converted one-by-one to JSON. Under the hood is uses Pandoc.

It receives a list of extensions that are instantiated and notified upon certain events. The events are documented in *AbstractExtension*.

class innoconv.runner.**InnoconvRunner**(*source_dir*, *output_dir*, *manifest*, *extensions*)
Convert content files in a directory tree.

run()
Start the conversion by iterating over language folders.

1.5.12 innoconv.utils

Utility module.

innoconv.utils.**to_ast**(*filepath*)
Convert a file to abstract syntax tree using pandoc.

Parameters **filepath** (*str*) – Path of file

Return type (list of dicts, str)

Returns (Pandoc AST, title)

Raises

- **RuntimeError** – if pandoc exits with an error
- **ValueError** – if no title was found

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

Symbols

A

C

D

E

F

G

H

I

J

M

P

R

S

T

W