# Table of contents

Chapter 1.

# Publishing your first book

**easybook** is an application that lets you easily publish books in various electronic formats. Although it was originally designed to publish programming books, you can use **easyboook** to publish any kind of book, manual or documentation website.

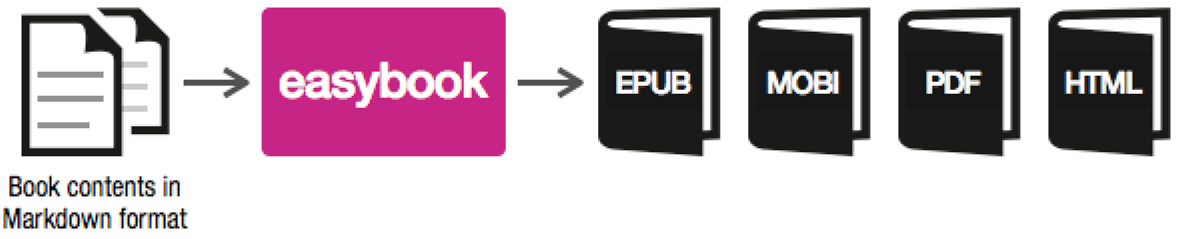


**Figure 1.1** easybook workflow diagram

**easybook** is currently designed for authors with advanced computer skills or companies with IT departments or personnel. For that reason, **easybook** lacks a GUI interface and it can only be used from the command line.

**easybook** is also a free and open-source application published under the MIT license (http://opensource.org/licenses/MIT) . This means that you can do *almost* anything with it. The only condition is that if you use **easybook** in your applications, you must always maintain the original `LICENSE.md` file included in the source code of **easybook**. This file explains the license of the application and its original author.

## 1.1. Installing easybook

Before installing **easybook**, make sure that you have PHP 5.3.2 or higher installed on your computer. This is the only technical requisite to install and use **easybook**. You can check your installed PHP version executing the following command on the console:

~~~ .cli $ php -v ~~~

The recommended method to install **easybook** is to use Composer (http://getcomposer.org/) :

~~~ .cli $ php composer.phar create-project easybook/easybook ~~~

Replace the `<installation-dir>` with the path where you want **easybook** to be installed and you are done. If Composer isn't installed on your computer, you can install it executing the following command:

~~~ .cli $ curl -s http://getcomposer.org/installer | php ~~~

Once installed, check that everything is correct by executing the `./book` script inside the **easybook** installation directory. If you see the list of available commands, everything went fine:

~~~ .cli $ cd $ ./book

```
|                         |
```

,---.,---.,---., .|---.,---.,---.|__/ |---',---|`---.|  ||  ||  ||  ||  \`---'`---^---'`---|---`
'`---'---' `---'

easybook is the easiest and fastest tool to generate technical documentation, books, manuals and websites.

Available commands: benchmark Benchmarks the performance of book publishing customize Eases the customization of the book design help Displays help for a command list Lists commands new Creates a new empty book publish Publishes an edition of a book version Shows installed easybook version ~~~

If the `./book` script doesn't work, try `php  book` or check the execution permissions of the `book` script.

The `./book` script is the unique *entry point* for every **easybook** command. If you need to know for example the installed version, just run the `version` command through the `book` script:

~~~ .cli $ ./book version

> | |

,---.,---.,---., .|---.,---.,---.|__/ |---',---|---- . | || || || || \---'----^---'----|---
'---'---' `---'

easybook installed version: 5.0 ~~~

### 1.1.1. Alternative installation methods

If you just want to test drive **easybook**, download the easybook.zip (https://github.com/javiereguiluz/easybook-package/blob/master/easy-book.zip?raw=true) file, uncompress it somewhere and you are done.

If you are an advanced user that want to *hack* or modify **easybook**, clone its repository with Git and install its dependencies with Composer:

~~~ .cli $ mkdir easybook $ git clone http://github.com/javiereguiluz/easybook.git easybook

// download vendors and dependencies $ cd easybook $ php composer.phar install ~~~

## 1.2. Publishing the first book

Before creating your first book in the next section, let's dig into book publication using the **easybook** documentation as the sample book. This book is located at the `doc/easybook-doc-en/` directory and it's a good resource for learning how to create advanced books with **easybook**. (#creating-a-new-book)

In order to transform the original Markdown contents of the documentation into a real book, execute the following command:

---

~~~ .cli $ cd $ ./book publish easybook-doc-en web ~~~

---

The first argument of the `publish` command is the name of the directory where the book contents are stored (`easybook-doc-en`) and the second argument is the name of the `edition` to be published. After executing this command, you should see the following output:

---
~~~

~~~ .cli $ ./book publish easybook-doc-en web

```
|                |
```

```
,---.,---.,---., .|---.,---.,---.|__/ |---',---|---. |  ||  ||  ||  ||  \---'---^---'---|---
'---'---' `---'
```

Publishing 'web' edition of 'easybook documentation' book...

[ OK ] You can access the book in the following directory: /doc/easybook-doc-en/Output/web

The publishing process took 0.5 seconds ~~~

(#creating-a-new-book)

The `publish` command always displays the edition (`web`) and the title (`easybook documentation`) of the book being published. Editions are one of the most powerful **easybook** features and they are thoroughly explained in (#creating-a-new-book) their own chapter. For now, consider that a single book can be published in a lot of different ways and formats, each of one being an edition. (#editions)

In case of error, the output of the `publish` command also displays the error cause and most of the times,how to solve it. If everything went fine, the `publish` command displays the path of the directory where that book edition was published (`<installation-dir>/doc/easybook-doc-en/Output/web`) and the total time elapsed to publish the book (`0.5 seconds`).

Browse to the `<installation-dir>/doc/easybook-doc-en/Output/web` directory and you'll find an HTML file called `book.html`. If you open it with a browser, you'll see the full **easybook** documentation published as a single HTML page.

Books published as single HTML pages aren't very useful, so execute now the following command to publish the book as a website:
~~~

~~~ .cli $ ./book publish easybook-doc-en website ~~~

If you browse to the `<installation-dir>/doc/easybook-doc-en/Output/website` directory, you'll find a directory called `book/` with a lot of HTML files inside it. As you can see, just by publishing another edition (`website` instead of `web`) the same book is published in a totally different way.

Execute once again the `publish` command but using the `ebook` edition:

~~~ .cli $ ./book publish easybook-doc-en ebook ~~~

As you surely have guessed, the book is now published as a `book.epub` file inside the `<installation-dir>/doc/easybook-doc-en/Output/ebook` directory. Now you can copy this `book.epub` file into any `.ePub` compatible reader (iPad tablets, iPhone phones, most Android tablets and phones and every e-book reader except Amazon Kindle) and start reading the **easybook** documentation as an e-book.

Similarly, you can publish your book as `MOBI` (the format for Kindle- compatible e-books) and `PDF` files. However, these formats require the installation of two libraries and they will be explained in the next chapters:

~~~ .cli // publishes the book as a PDF file $ ./book publish easybook-doc-en print

// publishes the book as a Kindle-compatible MOBI e-book $ ./book publish easybook-doc-en kindle ~~~

(#editions)

For those impatient readers that cannot wait to try the `MOBI` and `PDF` publication, these are the third-party libraries needed to generate those formats:

(#editions)

(#editions)
~~~

- For `MOBI` books: (#editions) KindleGen (http://amzn.to/kindlegen)
- For `PDF` books: PrinceXML (http://www.princexml.com/)

## 1.3. Creating a new book

In this section you will create and publish your own book. **easybook** requires that your books follow a certain structure of files and directories. To avoid creating this structure by hand, books are bootstrapped with the `new` command:

~~~ .cli $ ./book new "My First Book" ~~~

The only argument required by the `new` command is the title of the book enclosed with quotes. After executing the previous command, **easybook** will create a new directory called `my-first-book` inside the **easybook** `doc/` directory, and with the following file and directory structure:

~~~ / doc/ my-first-book/ config.yml Contents/ chapter1.md chapter2.md images/ Output/ ~~~

These are the files and directories created by **easybook**:

- `config.yml`, this file contains all the book configuration options. We'll cover all of them in the next chapters, but meanwhile you can change the `author` option to set your name as the book author's name.
- `Contents/`, this directory stores all the book contents (both text and images). **easybook** creates two sample chapters (`chapter1.md` and `chapter2.md`) and an empty `images/` directory.
- `Output/`, initially this directory is empty, but eventually it will contain the published book.

### 1.3.1. Writing the book

Book contents are written using regular text files with the Markdown syntax. This format has become the *de facto* standard for writing documentation for Internet. If you are not familiar with the Markdown syntax, read the Markdown reference appendix. (#markdown-reference)

Markdown is the only format currently supported by **easybook**. In the future, more formats will be supported, such as reStructuredText and Textile.

---

Therefore, forget **easybook** for a while and write the contents of your book in

---

Markdown syntax using your favorite text editor (`vi`, *Notepad*, *TextMate*, *SublimeText*, etc.)

---

The `config.yml` file lists all the book items, such as the chapters and appendices. The sample book created with the `new` command only includes two chapters. Therefore, whenever you add a new chapter to the book, don't forget to add it to the `config.yml` file too:

---

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: 1, content: chapter1.md } - { element: chapter, number: 2, content: chapter2.md } - { element: chapter, number: 3, content: chapter3.md } - { element: chapter, number: 4, content: chapter4.md } - { element: chapter, number: 5, content: chapter5.md } - ... ~~~

---

(#markdown-reference)

The next chapter explains in detail all the configuration options of this file, and the 21 different (#markdown-reference) content types that **easybook** books can include. (#content-types)

As the book publication process is extremely fast (less than 2 seconds for a 400-page book on an average computer), you can publish every edition of the book periodically to check your progress:

---

~~~ .cli $ ./book publish my-first-book web $ ./book publish my-first-book website $ ./book publish my-first-book ebook $ ./book publish my-first-book kindle $ ./book publish my-first-book print ~~~

---

(#content-types)

(#content-types)

Chapter 2.

# Book contents and configuration

This chapter explains all the available book configuration options and all the content types defined by **easybook**.

## 2.1. Book configuration

All the book configuration is centralized in a single file called `config.yml` at the root directory of the book. Usually this configuration file is divided into three parts:

~~~ .yaml book: # FIRST part: basic book information title: "..." author: "..." edition: "..." language: "..." publication_date: "..."

```
# SECOND part: book contents
contents:
    - ...
    - ...
    - ...

# THIRD part: book editions
editions:
    edition1:
        # ...
```
~~~

```
    edition2:
        # ...
    # ...
```

~~~

The first part of this file sets the basic book information (enclose the value of each option with single or double quotes):

~~~ .yaml book: title: "The Origin of Species" author: "Charles Darwin" edition: "First edition" language: en publication_date: "1859-11-24" ~~~

- `title`, sets the title of the book. By default **easybook** uses the title that you typed when creating the book with the `new` command, but you can change it if needed.
- `author`, sets the author of the book. If the book has more than one author, separate them with commas (e.g. `James Watson, Francis Crick`).
- `edition`, sets the name of the current book edition. This is considered as the traditional *"literary edition"* and has nothing to do with the edition concept explained in the next chapter.
- `language`, sets the language of the book contents with a two letter code (`en` for English, `es` for Spanish, `fr` for French, etc.) This value is used to generate several elements of the book, such as the labels and the default titles. If the book is written in several languages, set the code of the main book language.
- `publication_date`, sets the publication of the book in `YYYY-MM-DD` format. If you set this value as null (`publication_date: ~`) **easybook** will automatically set the publication date to the day on which the book is published.

The `contents` and `editions` options are explained in detail in the next section and in the next chapter respectively. The `contents` option defines the book contents and their order. The `editions` option defines the features of each edition of the book.

## 2.2. Content types

Books list their contents with the `contents` option in the `config.yml` file. After creating a new book with the `new` command, its default contents are:

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: 1, content: chapter1.md } - { element: chapter, number: 2, content: chapter2.md } ~~~

(#content-types)

The most important option of each content is `element`, which defines its content type. **easybook** currently supports 21 content types (the following definitions have been (#content-types) extracted from the Wikipedia (http://en.wikipedia.org/wiki/Book_design) :

- `acknowledgement`, often part of the preface, rather than a separate section in its own right. It acknowledges those who contributed to the creation of the book.
- `afterword`, a piece of writing describing a time well after the time frame of the main story of the book.
- `appendix`, is a supplemental addition to a given main work. It may correct errors, explain inconsistencies or otherwise detail or update the information found in the main content of the book.
- `author`, information about the book author/authors.
- `chapter`, the most used type in regular books.
- `conclusion`, the end of the book or document, where all of the pending issues are resolved or where idea and thoughts are settled.
- `cover`, the cover of your book.
- `dedication`, a page that usually precedes the text, in which the author names the person or people for whom he has written the book.
- `edition`, information about the current edition of the book, including the publication date.
- `epilogue`, a piece of writing at the end of a work of literature or drama, usually used to bring closure to the work.
- `foreword`, usually written by any person other than the author of the book. It often tells of some interaction between the writer of the foreword
~~~

and the story or the writer of the story.

- `glossary`, consists of a set of definitions of words of importance to the work, normally alphabetized.
- `introduction`, a beginning section which states the purpose and goals of the following writing.
- `license`, information about the copyright holder or any other author or publisher rights that affect the book.
- `lof` (*list of figures*), is an ordered list of the book images and illustrations, including their numbers and captions. The page in which the image is included is also shown for `pdf` type editions.
- `lot` (*list of tables*), is an ordered list of the book data tables, including their numbers and captions. The page in which the table is included is also shown for `pdf` type editions.
- `part`, used to group several related chapters or appendices.
- `preface`, generally covers the story of how the book came into being, or how the idea for the book was developed.
- `prologue`, *written* by the narrator or any other character in the book. It's an opening to a story that establishes the setting and gives background details, often some earlier story that ties into the main one, and other miscellaneous information.
- `title`, the page at or near the front which displays book title and author, usually together with information relating to the publication of the book.
- `toc` (*table of contents*), a list of the parts of a book or document organized in the order in which the parts appear.

These 21 content types defined by **easybook** are enough to publish most books, but if you need it, you can also define your own new content types. (#new-content-types)

Some book items don't require any other option besides `element`:

---

~~~ .yaml book: # ... contents: - { element: cover } - { element: title } - { element: license } - { element: toc } - { element: chapter, number: 1, content: chapter1.md } - { element: chapter, number: 2, content: chapter2.md } - ... ~~~

---
~~~

In any case, each book item can define the following three options:

- `number`, the number of the item used to generate the labels of each section heading (`1.1`, `1.2`, `1.2.1`, `1.2.2`, etc.). It's mostly used for chapters and appendices and **easybook** doesn't restrict its format, so you can use Roman numerals (`I.1`, `I.2`), letters (`A.1`, `A.2`) or any other symbol or string.
- `content`, the name of the file with the contents of this item. The file name should include the extension (`.md` in the case of Markdown). The value of this option is interpreted as the relative path from your book `Contents/` directory, so you can add all the subdirectories you want.
- `title`, the title of the item. In the case of chapters and appendices, it's not necessary to use it, because **easybook** extracts automatically their titles from the first `<h1>` of the chapter/appendix content. For the rest of content types, it's also unnecessary to set the title because the default title is usually enough (and it's displayed in the same language of the book contents). In sum, this option is only used for `part` content type, which separates the book contents into parts or sections.

## 2.3. Defining book contents

The structure of a book can be very complex (cover, title page, acknowledgements, dedication, chapters, parts, etc.). **easybook** supports all the common book content types, but for now, we'll just focus on the chapters. You can add as many chapters as you want and each one can be as large or as short as you need. All you have to do is to list the book chapters under the `contents` option of the `config.yml` file:

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: 1, content: chapter1.md } - { element: chapter, number: 2, content: chapter2.md } ~~~

Each line under the `contents` option defines a content of the book. Add your book chapters as `chapter` elements and set their number (with the `number` option) and the name of the files that hold their contents (with the `content` option).
~~~

Besides being lightning-fast, the main feature of **easybook** is its flexibility, as it never forces you to work in a certain way. Do you want to number your chapters in an *imaginative* way? People will think you're crazy, but **easybook** allows you to do it:

---

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: 100, content: chapter1.md } - { element: chapter, number: 56, content: chapter2.md } ~~~

---

Do you need letters instead of numbers? This is most appropriate for appendices instead of chapters, but **easybook** won't stop you from doing it:

---

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: A, content: chapter1.md } - { element: chapter, number: B, content: chapter2.md } ~~~

---

A recommended best practice is to use long and semantic names for book content files:

---

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: 1, content: 01-publishing-your-first-book.md } - { element: chapter, number: 2, content: 02-book-contents-and-configuration.md } ~~~

---

The most important thing about the `contents` option is the order in which you list the book items. The published book will be always composed of those contents and in that order. Therefore, the following configuration will output a book with the cover between the two chapters and the table of contents at the very end (it's completely crazy, but **easybook** allows you to do *almost* anything):

---

~~~ .yaml book: # ... contents: - { element: chapter, number: 1, content: 01-publishing-your-first-book.md } - { element: cover } - { element: chapter, number: 2, content: 02-book-contents-and-configuration.md } - { element: toc } ~~~

---

By default, all content files are stored in the `Contents/` directory. However, if your book is complex, you can divide the contents into several subdirectories. Then, include the subdirectory in the `content` file path (don't forget to enclose it with quotes if the file path has white spaces):

---

~~~ .yaml book: # ... contents: - { element: cover } - { element: toc } - { element: chapter, number: 1, content: introduction/chapter1.md } - { element: chapter, number: 2, content: introduction/chapter2.md } - { element: chapter, number: 3, content: advanced/chapter1.md } - { element: appendix, number: A, content: appendices/appendixA.md } ~~~

---

The above configuration means that your book contents are distributed this way:

---

~~~ / ... Contents/ introduction/ chapter1.md chapter2.md advanced/ chapter1.md appendices/ appendixA.md ~~~

---

## 2.3.1. Different directories per book

Unless stated otherwise, the books are created in the `doc/` directory of the **easy-book** installation directory. If you want to save the contents in any other directory, add the `--dir` option when creating and/or publishig the book:

---

~~~ .cli $ ./book new --dir="/Users/javier/books" "My First Book" // the book is created in "/Users/javier/books/my-first-book"

---

$ ./book publish --dir="/Users/javier/books" my-first-book print // the book is published in // "/Users/javier/books/my-first-book/Output/print/book.pdf" ~~~

---

(#new-content-types)
~~~

(#new-content-types)

Chapter 3.

# Editions

**easybook** allows you to publish the very same book in radically different ways. This is possible thanks to the **editions**, that define the specific characteristics of the published book.

Editions are defined under the `editions` option in the `config.yml` file. By default, the books created with the `new` command define five editions named `ebook`, `kindle`, `print`, `web` and `website`:

~~~ .yaml book: # ... editions: ebook: format: epub # ...

```
     kindle:
         extends: ebook
         format:  mobi

     print:
         format:  pdf
         # ...

     web:
         format:  html
         # ...

     website:
         extends: web
```
~~~

```
          format:  html_chunked
```

~~~

## 3.1. Edition formats

A single book can define as many editions as needed. The only requirement is that the name of each edition must be unique for the same book and cannot contain white spaces.

Each edition is published at the `Output/` directory of the book, in a subdirectory named after the edition name. You can define any number of editions, but all of them must belong to one of the following five types defined by the `format` option:

- `epub`, the book is published as an e-book named `book.epub`.
- `mobi`, the book is published as a Kindle-compatible e-book named `book.mobi`.
- `pdf`, the book is published as a PDF file named `book.pdf`.
- `html`, the book is published as a HTML page named `book.html`.
- `html_chunked`, the book is published as a static website in a directory named `book/`.

## 3.2. Configuration options

The purpose of the editions is to define a set of unique characteristics for the published book. This is done with the several configuration options defined by **easybook** for the editions. Some of these options are common for every edition format and others are specific to each format.

### 3.2.1. Common configuration options

- `highlight_code`, this option is only useful for technical books that include code listings. If `true`, the syntax of the code is highlighted.
- `highlight_cache`, this option is only useful for technical books that
~~~

include source code. If `true`, all the highlighted code listings will be cached to boost book publishing performance. By default this cache is disabled because it's only appropriate for complex books that are generated regularly.

* `include_styles`, if `true` **easybook** default CSS styles are applied to the published book. If you want to fully customize the design of your book, don't apply these styles. However, most of the time it's better to apply thee default styles and tweak the design with the `customize` command.
* `labels`, sets the content types for which **easybook** will add labels to their section headings. By default labels are only added to headings of chapters and appendices. In addition to the regular content types, you can use two special values called `figure` and `table` to add labels for book images and tables. If you don't want to show labels in your book, set an empty value for this option: `labels: ~`.
* `toc`, sets the options of the table of contents. It's ignored unless the book has at least one `toc` element type. It has two options:
    * `deep`, the maximum heading level included in the TOC (`1` is the lowest possible number and would only show `<h1>` level headings; `6` is the highest possible value and would show all `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` and `<h6>` headings).
    * `elements`, the type of elements included in the TOC (by default, only `appendix`, `chapter` and `part` are included).

### 3.2.2. Format specific configuration options

In addition to the common configuration options, some formats define their own specific configuration options:

(#new-content-types)

(#new-content-types)

* (#new-content-types) Configuration options for the `html_chunked` format (#html-configuration-options)

  (#html-configuration-options)

* (#html-configuration-options) Configuration options for the `pdf` format (#pdf-configuration-options)

  (#pdf-configuration-options)

## 3.3. Before and after scripts

Publishing a digital book usually involves more than the actual publication of the book. Sometimes, before publishing the book, its contents must be updated by downloading them from a remote server. After the publication, you may need to copy the book into another directory or you may have to notify some other system about the publication.

(#pdf-configuration-options)

**easybook** simplifies all these tasks with the `before_publish` and `after_publish` edition options. These options are modeled after the popular (#pdf-configuration-options) `before_script` and `after_script` Travis CI options (http://about.travis-ci.org/docs/user/build-configuration/#before_script%2C-after_script) and allow you to execute commands before or after the book publication without the need to define a custom plugin or a script.

Each option admit an array of commands that are executed sequentially:

~~~ .yaml book: title: '...' # ...

```
editions:
    # ...

    website:
        format:      html_chunked
        chunk_level: 2
        # ...

        before_publish:
            - echo "This command is executed before book
publishing"
            - git pull ...
            - cp ...

        after_publish:
            - "/home/user/scripts/notify_book_publish.s
h"
            - "/home/user/scripts/update_google_sitema
p_xml.sh"
```

~~~

If you only need to execute one command, you can replace the array for a simple string:

~~~ .yaml book: # ...

```
editions:

    website:
        # ...
        before_publish: "cd ... && git pull"
        after_publish:  "/home/user/scripts/notify_boo
k_publish.sh"
```

~~~

As any other **easybook** command, all of these scripts are rendered as Twig strings, so they can easily access to any application property:

~~~ .yaml book: # ...

```
editions:

    website:
        # ...
        before_publish: "git clone git://github.com/{{ a
pp.get('publishing.book.slug') }}/book"
        after_publish:  "cp ... /home/{{ book.author
}}/books/{{ 'now'|date('YmdHis') }}_book.pdf"
```

~~~

## 3.4. Extending editions

In addition to the previous configuration options, editions can also set a very useful option named `extends`. The value of this option is interpreted as the name of the edition from which this edition *inherits*. When an edition *inherits* from another edition, the options of the parent edition are copied on the *heir* edition, which can then override any value.

Imagine for example that you want to publish one PDF book with three slightly different designs. The draft version (`draft`) must be double-sided and must have very small margins to reduce its length, the normal version (`print`) is one-sided and has normal margins. The version prepared for lulu.com website (`lulu.com`) is similar to the normal version, except is double-sided:

~~~ .yaml book: # ... editions: print: format: pdf margin: top: 25mm bottom: 25mm inner: 30mm outter: 20mm page_size: A4 two_sided: false

```
    draft:
        extends:       print
        margin:
            top:       15mm
            bottom:    15mm
            inner:     20mm
            outter:    10mm
        two_sided:     true

    lulu.com:
        extends:       print
        two_sided:     true
```

~~~

First, define a new `pdf` format edition called `print` and set the page size, the margins and disable the two-sided printing. Then, instead of creating a new edition for the `draft` book, you inherit from the previous edition (`extends: print`) and override its margins and the `two_sided` option.

Similarly, as the `lulu.com` version is almost the same as the `print` edition, instead of creating a new edition, you just inherit from it (`extends: print`) and then you override the only different configuration option.

The only limitation of `extends` is that it only works with one level of inheritance. Therefore, and edition cannot extend another edition that extends a third one.

If several editions share a large set of configuration options, you can group them under a common *fake* edition and make the other *real* editions extend from it.

If for example your `before_publish` and `after_publish` scripts are the same for every edition, group them in a *fake* edition:

~~~ .yaml book: # ... editions: common: before_publish: ... after_publish: ...

```
     edition1:
```
~~~

```
      extends: common
      # ...

   edition2:
      extends: common
      # ...

   edition3:
      extends: common
      # ...
```

~~~
~~~

Chapter 4.

# Themes

A theme is a set of templates, stylesheets and other resources that define the visual design of the book. **easybook** already includes a theme for each edition type (`epub`, `mobi`, `pdf`, `html`, `html_chunked`), so your books will look professional without any effort.

**easybook** themes are located in the `app/Resources/Themes/` directory. If you need to change the design of your books, **don't** modify those files, but use the overriding techniques explained in this chapter.

## 4.1. Contents

### 4.1.1. Default contents

In most books, the only elements that define their own content are chapters and appendices (with the `content` option). For that reason, **easybook** defines sensible default contents for some content types. If your book for example includes a `license` content type without any content:

~~~ .yaml book: # ... contents: - ... - { element: license } - { element: chapter, content: 'chapter1.md' } - ... ~~~

**easybook** will use the following as the content for this element:

~~~ .twig © Copyright {{ book.publication_date|default('now')|date('Y') }}, {{ book.author }}

**ALL RIGHTS RESERVED**. This book contains material protected under International and Federal Copyright Laws and Treaties. Any unauthorized reprint or use of this material is prohibited. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system without express written permission from the author and/or publisher. ~~~

The default `license` page displays a copyright notice corresponding to either the book publication date (if defined) or the current year. Similarly, the `title` content type defines the following default content:

~~~ .twig

# {{ book.title }}

## {{ book.author }}

### {{ book.edition }}

~~~

The contents defined by **easybook** depend on both the edition being published and the content type. You can access these default contents at the `Contents/` directory of the theme.

### 4.1.2. Custom contents

If you don't want to use **easybook** default contents for some element, simply add the `content` option indicating the file that defines its contents:

~~~ .yaml book: # ... contents: - ... - { element: license, content: creative-commons.md } - { element: title, content: my-own-title-page.md } - ... ~~~
~~~

## 4.2. Templates

### 4.2.1. Default templates

The content of each element is *decorated* with a template before including it in the published book. **easybook** templates are created with Twig (http://twig.sensiolabs.org/) , the best templating language for PHP. You can access all the default templates in the `Templates/` directory of the theme.

This is for example the template used to decorate each chapter of a PDF book:

~~~ .twig
~~~

{{ item.label }}

# {{ item.title }}

> `{{ item.content }}`

\~\~\~

The data of the book item being decorated is accessible through a special variable called `item`, which stores the following properties:

* `item.title`, is the title of the book item. By order of priority, this value is obtained from 1) the `title` configuration option of the item, 2) the `<h1>` title of the item content, 3) the default title defined by **easybook** for this kind of element.
* `item.slug`, is a *safe* version of the `title` that doesn't include white spaces or any other *troublesome characters* (such as accents, `.`, `?`, `!`, ...). This value is used on URLs, on `id` HTML attributes, etc.
* `item.label`, the label of the main heading of the item. Usually it's only defined for chapters (`Chapter XX`), parts (`Part XX`) and appendices (`Apendix XX`).
* `toc`, array with the table of contents of this item. It's empty for most of the content types (`cover`, `license`, etc.) but can be very large for complex chapters and appendices.
* `item.content`, the content of the item prepared to be included in the book (it has already been converted from its original Markdown format).
* `item.original`, the original content of the item without any modification.

- `item.config`, array with all the configuration options defined for the item in the `config.yml` file. Internally it stores the `number`, `title`, `content` and `element` properties. For example, you can access the type of the item with the following expression: `{{ item.config.element }}`.

Images and tables are decorated with special templates called `figure.twig` and `table.twig` which have access to the following variables:

- `item.caption`, is the title of the image/table as indicated in the original content.
- `item.content`, is the complete HTML code generated to display the image/ table in the book (`<img src="..." alt="..." />` or `<table> ... </ table>`).
- `item.label`, is the label of the image/table. This value is empty unless the `labels` option of the edition includes `figure` and/or `table`.
- `item.number`, is the autogenerated number of the image/table inside the item being processed. The first image/table is considered to be the number `1` and the following are increased by one.
- `element.number`, is the number of the parent element (chapter, appendix, etc.) of the image/table. This property will be for example `5` for all the images/tables included in the chapter number `5` of the book.

In addition to these properties, all the **easybook** templates have access to three global variables:

- `book`, provides direct access to the configuration options defined under `book` in `config.yml` file. You can access for example the book author in any template using `{{ book.author }}` and the current **easybook** version using `{{ book.generator.version }}`.
- `edition`, provides direct access to the configuration options of the edition currently being published.
- `app`, provides direct access to all the configuration options and services of **easybook** (defined in the `src/DependencyInjection/Application.php` file).

### 4.2.2. Custom templates

If you want to fine-tune the appearance of your book, you can start by using your own templates instead the default ones. First, create a directory called `Resources/` within the directory of your book and inside it, create another directory called `Templates/`.

You can now define your own templates in the `Resources/Templates/` to apply them to the published book. The name of the template must match the item type (`chapter`, `dedication`, `author`, etc..) and its extension should be `.twig` because they are always created with the Twig templating language. Consequently, if you want to modify the design of the chapters, you must create a template called `chapter.twig` in the `<book>/Resources/Templates/` directory.

When a book is published, each book item is decorated with its own template. **easybook** looks for each template in the following directories (showed in order of priority). If none of these templates are found, it will use the default template:

1. `<book>/Resources/Templates/<edition-name>/template.twig`, allows you to modify the design for each edition. The directory inside `Templates/` must match exactly the name of the edition being published.
2. `<book>/Resources/Templates/<edition-type>/template.twig`, allows you to modify the design for all the editions of the same format. The directory inside `Templates/` must be named `epub`, `mobi`, `html`, `html_chunked` or `pdf`.
3. `<book>/Resources/Templates/template.twig`, applies the same same design to all editions. The template is located in the `Resources/Templates` directory without including it in any subdirectory. This option is rarely used because it usually doesn't make sense to apply the same style regardless of whether content is converted into a PDF file or a website.

As explained in the previous section, the templates have access to a variable named `item` with all the information about the item that is being decorated and three global variables with information about the book (`book`), the edition being published (`edition`) and the entire application (`app`). (#default-templates)

In addition to the templates associated with the book item (`chapter.twig`, etc.) each edition format defines several special templates, as explained in the following chapters.

## 4.3. Styles

**easybook** uses CSS stylesheets to define the visual design of the books, even for the `epub` and `pdf` type editions.

### 4.3.1. Default styles

**easybook** applies by default some styling to the book that depends on the selected `theme` and the edition being published. If you want to disable these styles for a specific edition, set its `include_styles` option to `false` and no default style will be applied to the book.

The CSS styles for each format are defined in the `styles.css.twig` file located at the `<theme>/<edition-format>/Templates/` directory of the theme. The CSS styles applied to the book vary greatly depending on the format of the published edition:

(#default-templates)

* `epub` and `mobi`: CSS styling is pretty limited due to the abysmal support of most e-book readers.
* `html` and `html_chunked`: the only limit is the highest CSS version that you can use on your website (CSS 2.1, CSS 3, CSS 4, etc..) because every modern web browser offers an excellent CSS support.

  (#default-templates)
* `pdf`: your imagination is the limit. PDF books are generated by (#default-templates) PrinceXML (http://www.princexml.com/) , which defines tens of new CSS properties and options unavailable in the CSS standards.

### 4.3.2. Custom styles

In addition to the **easybook** default styles, your book can define its own CSS styles to tweak its design. Unless you need a complete redesign of the book, it's recommended to maintain the default **easybook** styles (`include_styles` option set to `true`) and then apply your custom CSS to include new styles or modify them. To do this, add a file called `style.css` within any of these directories:

1. `<book>/Resources/Templates/<edition-name>/style.css`, if you want to apply the styles just to one specific edition.

2. `<book>/Resources/Templates/<edtion-type>/style.css`, if you want to apply the same styles to all the editions of the same type (`epub`, `mobi`, `html`, `html_chunked` or `pdf`).
3. `<book>/Resources/Templates/style.css`, if you want to apply the same styles to all the editions of the book.

Instead of creating this CSS file by hand, **easybook** includes a command called `customize` that generates the needed CSS for each edition. The first argument of the command is the book directory and the second argument is the name of the edition to be customized:

~~~ .cli $ ./book customize my-book ebook

OK: You can now customize the book design with the following stylesheet: /doc/my-book/Resources/Templates/ebook/style.css ~~~

The CSS file generated with the `customize` command is different for every edition format (`html`, `pdf`, `epub`, `mobi`) and it includes some comments to ease the customization of the book design.

## 4.4. Fonts

### 4.4.1. Default fonts

**easybook** bundles several free and high-quality fonts to make published books look professional. You can see the fonts and their license in the `app/Resources/Fonts/` directory.

## 4.5. Labels and titles

### 4.5.1. Default labels and titles

**easybook** requires each content to have its own **title**. Define the title of each item with the `title` configuration option:

~~~ .yaml book: # ... contents: - ... - { element: part, title: 'Introduction' } - { element: chapter, number: 1, content: chapter1.md } - { element: chapter, number: 2, content: chapter2.md } - { element: part, title: 'Advanced Concepts' } - { element: chapter, number: 3, content: chapter3.md } - ... ~~~

Chapters and appendices don't have to explicitly set their title in the `config.yml` file. The reason is that **easybook** looks for the first `<h1>` heading of the chapter or appendix and considers it as the title.

For the rest of items, if you don't set explicitly their titles, **easybook** assigns them a default title that varies depending on the language in which the book is written. The following are for example the default titles applied to English books (as defined in the `app/Resources/Translations/titles.en.yml` file):

~~~ .yaml title: acknowledgement: 'Acknowledgements' afterword: 'Afterword' author: 'About the author' conclusion: 'Conclusion' cover: 'Cover' dedication: 'Dedication' edition: 'About this edition' epilogue: 'Epilogue' foreword: 'Foreword' glossary: 'Glossary' introduction: 'Introduction' license: 'License' lof: 'List of Figures' lot: 'List of Tables' preface: 'Preface' prologue: 'Prologue' title: 'Title' toc: 'Table of contents' ~~~

Moreover, the book can add **labels** (`Chapter XX`, `Appendix XX`, etc.) to the section titles using the `labels` option. As explained before, this option indicates the content types for which labels are added automatically.

Unless you modify this option, books only add labels in chapters and appendices. The default labels also depend on the language in which the book is written. The labels applied for English books are defined in the `app/Resources/Translations/labels.en.yml` file.

Unlike the titles, labels can contain variable parts, such as the appendix or chapter number. Therefore, **easybook** uses Twig templates to define each label:

~~~ .yaml label: figure: 'Figure {{ element.number }}.{{ item.number }}' part: 'Part {{ item.number }}' table: 'Table {{ element.number }}.{{ item.number }}' ~~~

The `figure` and `table` labels can use the same variables as the `figure.twig` and `table.twig` templates explained previously in the default templates section. Therefore, `{{ item.number }}` shows the autogenerated number of the image/table and `{{ element.number }}` shows the number of the chapter or appendix. (#default-templates)

Chapters and appendices must define six different labels, each corresponding to one of the six heading levels (`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` and `<h6>`). In the following example, appendices only include a label in their titles, thus leaving empty the last five levels:

~~~ .yaml label: appendix: - 'Appendix {{ item.number }}' - '' - '' - '' - '' - '' ~~~
~~~

Labels can access to all configuration options defined by the item in the `con-fig.yml` file. The label of a chapter can then use `{{ item.number }}` to show the number of the chapter. In addition to these properties, label templates have access to two special variables called `level` and `counters`.

---

The `level` variable indicates the section heading level for which you want to get the label, being `1` the level of `<h1>` heading and `6` the level of `<h6>` headings. The `counters` variable is an array with the counters of all heading levels. For that reason, to show second-level headings as `1.1`, `1.2` ... `7.1`, `7.2` you can use the following expression:

---

(#default-templates)

~~~ .yaml label: chapter: - 'Chapter {{ item.number }}' - '{{ item.counters[0] }}.{{ item.counters (#default-templates) 1 (http://twig.sensiolabs.org/) }}' - ... ~~~

Extending the previous example, you can use the following templates to format all the heading levels as `1.1`, `1.1.1`, `1.1.1.1`, etc.:

~~~ .yaml label: chapter: - 'Chapter {{ item.number }} ' - '{{ item.counters[0:2]|join(".") }}' # 1.1 - '{{ item.counters[0:3]|join(".") }}' # 1.1.1 - '{{ item.counters[0:4]|join(".") }}' # 1.1.1.1 - '{{ item.counters[0:5]|join(".") }}' # 1.1.1.1.1 - '{{ item.counters[0:6]|join(".") }}' # 1.1.1.1.1.1 ~~~

This last example clearly shows what you can achieve by combining the flexibility of **easybook** and the power of Twig.

### 4.5.2. Custom labels and titles

If you want to use your own titles and labels, you must first create the `Trans-lations` directory inside the `Resources` directory of your book (you must also create the latter if it doesn't exist). Then add the new label file in one of the following directories:

1. `<book>/Resources/Translations/<edition-name>/labels.en.yml`, if you want to change the labels in a single edition. The subdirectory of `Translations` must be named exactly like the edition being published.
2. `<book>/Resources/Translations/<edition-type>/labels.en.yml`, if you want to change the labels in all editions

of the same type. The directory in `Translations` can only be named `epub`, `mobi`, `pdf`, `html`, or `html_chunked`.

3. `<book>/Resources/Translations/labels.en.yml`, if you want to change the labels on all the editions of the book, regardless of its name or type.

When you use your own label file, you don't have to define the value of all labels. Add only the labels that you want to modify and **easybook** will assign to the rest their default values. Therefore, to only modify the label of the images in any book edition, create a new `<book>/Resources/Translations/labels.en.yml` file and add just the following content:

~~~ .yaml label: figure: 'Illustration {{ item.number }}' ~~~

If you want to change the titles instead of the labels, follow the same steps but create a file called `titles.en.yml` instead of `labels.en.yml`. If your book isn't written in English, replace `en` for the code of the other language (e.g. `labels.es.yml` for Spanish labels).
~~~

Chapter 5.

# Publishing HTML books

HTML books can be published either as a single HTML page (`html` format) or as a complete website (`html_chunked` format). Besides the common templates and configuration options explained previously, this chapter explains the custom templates and configuration options that HTML books can use.

## 5.1. Templates

The following list shows all the templates used by `html` format editions to publish the book. These are also the templates that you can override using your own templates: (#custom-templates)

- A template for each of the 21 content types: `acknowledgement.twig`, `afterword.twig`, `appendix.twig`, `author.twig`, `chapter.twig`, `conclusion.twig`, `cover.twig`, `dedication.twig`, `edition.twig`, `epilogue.twig`, `foreword.twig`, `glossary.twig`, `introduction.twig`, `license.twig`, `lof` (list of figures), `lot` (list of tables), `part.twig`, `preface.twig`, `prologue.twig`, `title.twig` and `toc.twig` (table of contents).
- `book.twig`, this is the template that ultimately assembles all the book contents.
- `code.twig`, the template used to decorate all the code listings included in the book.
- `figure.twig`, the template used to decorate all the images included in the book.

- `table.twig`, the template used to decorate all the tables included in the book.

(#custom-templates)

The `html_chunked` format editions don't use a different template for each content type. They only define the following five templates, which you can also override (#custom-templates) using your own templates: (#custom-templates)

- `code.twig`, the template used to decorate all the code listings included in the book.
- `figure.twig`, the template used to decorate all the images included in the book.
- `table.twig`, the template used to decorate all the tables included in the book.
- `index.twig`, the template that generates the front page of the website. It displayes the book title, the author name and the full table of contents.
- `chunk.twig`, generic template applied to all the the content types. This template must only include the contents of the item being decorated. The common features of the website are defined in the `layout.twig` template.
- `layout.twig`, generic template applied to all the content types, including the front page, after decorating them with their own templates. This is the best template to include the common features of the website, such as header, footer and links to CSS and JavaScript files.

## 5.2. Configuration options

Besides the common configuration options, the `html_chunked` editions define the following custom options:

- `chunk_level`, the level of the heading used to *chunk* or split the book into HTML pages. Its default value is `1`, meaning that each `<h1>` heading produce a new HTML page. In other words, the published website will have a single HTML page for each chapter, appendix, etc. When the book chapters are long, it's recommended to set the `chunk_level` option to `2`, meaning that each `<h1>` and `<h2>` will produce their own HTML page. In other words, each chapter section will be published in its own page, which is much better for long chapters. Currently, **easybook** only supports `1` and `2` level chunking.

- `images_base_dir`, the prefix used in every image URI. Its default value is `images/`, meaning that when you include an image in your book with `![...](image1.png)`, the published book will transform it to `<img src="images/image1.png" />`. When you embed a published book into another website, this path may no longer work, depending on your configuration. In those cases, use this option to set the correct prefix. For example, if you use `images_base_dir: '/img/doc/en'`, the previous image will be transformed into: `<img src="/img/doc/en/image1.png" />`

## 5.3. Syntax highlighting

(#custom-templates)

If you use **easybook** to write programming books or technical documentation, you can automatically highlight the code listings. There is nothing to install or configure, because **easybook** already includes the (#custom-templates) GeSHi library (http://qbnz.com/highlighter) to highlight the code listings. This library highlights more than 200 programming languages (`php`, `java`, `c`, `javascript`, `ruby`, `python`, `perl`, `erlang`, `haskell`, ...), markup languages (`html`, `yaml`, `xml`,...) and configuration files (`ini`, `apache`, ...).

### 5.3.1. Configuration

Syntax highlighting is disabled by default. Set the `highlight_code` option to `true` in any edition you want to highlight:

~~~ .yaml book: # ... editions: edition1: highlight_code: true # ...

```
        edition2:
            highlight_code: true
            # ...

        edition3:
            highlight_code: false
            # ...
```

~~~

## 5.3.2. Code block types

As developers cannot agree on what code block style should be used, **easybook** supports the three most used code block types: 1) Markdown classic, 2) Fenced, 3) GitHub.

**1)** This is the default type and it's based on the traditional 4-spaces or tab indentation of code blocks. The difference is that **easybook** augments this format to allow you set the programming language of the code with a special tag in the first line of the code listing:

~~~ [php] $lorem = 'ipsum'; // ... ~~~

When the first line of the code block is the name of a programming or configuration language enclosed with brackets (`[` and `]`), this special tag is striped from the listing and the rest of the code is appropriately highlighted.

If some code listing is language agnostic or you don't want to highlight it, don't include any language tag or if you prefer it, add the generic `[code]` tag:

~~~ [code] // This code won't be highlighted // ... ~~~

The problem with this code block type is that all the code must be indented and therefore, there cannot be empty lines without leading tabs or white spaces. This is a **huge problem** with editors that remove tabs and white spaces in empty lines.

This code will be wrongly parsed:

~~~ [php] $lorem = 'ipsum'; (no spaces or tabs in this line) $another_lipsum = 'ipsum'; // ... ~~~

The same code, with tabs or white spaces in every line, works perfectly:

~~~ [php] $lorem = 'ipsum'; (4 white spaces in this line) $another_lipsum = 'ipsum'; // ... ~~~

The other two code block types work perfectly whatever the code you write, so you don't have to mess around with leading tabs or white spaces.

**2)** `fenced`, this is the style defined by the PHP Markdown library (http://michelf.ca/projects/php-markdown/) :

- Define the start of the code block with at least three `~~~`
- Optionally set the programming language name, prefixing it with a dot.

- Include the code without any indentation.
- Define the end of the code block using the same number of `~~~` as the opening of the block.

Examples:

~~~ ~~~ .php $lorem = 'ipsum'; // ... ~~~ ~~~

~~~ ~~~~~~~~~~~~~~~~~~~~~~ .php $lorem = 'ipsum'; // ... ~~~~~~~~~~~~~~~~~~~~~~ ~~~

~~~ ~~~ // some generic code // without any programming language // ... ~~~ ~~~

**3)** `github`, this is the style used by GitHub and it's very similar to the fenced style. Instead of three tildes (`~~~`), use three backticks:

~~~ `php $lorem = 'ipsum'; // ...` ~~~

~~~ `// some generic code // without any programming language // ...` ~~~

You can use any code block type, but a given book can only use one type for all its code listings. The Markdown classic code block type is enabled by default. To enable one of the other two code block types, use the `code_block_type` global parameter:

~~~ .yaml easybook: parameters: parser.options: code_block_type: fenced # alternatively, you can also use: # code_block_type: github

book: title: ... # ... ~~~

### 5.3.3. Improve the syntax highlighting performance

Syntax highlighting is a very slow and time consuming process for books with hundreds to thousands lines of code. In order to improve its performance, **easybook** includes an internal highlight cache.

This cache is disabled by default, because it's not useful for books with a small number of lines of code. However, for large books it can reduce more than 90% the publication time.

Set the `highlight_cache` option to `true` in any edition you want to enable this cache:

~~~ .yaml book: # ... editions: edition1: highlight_code: true highlight_cache: true # ... ~~~
~~~

Chapter 6.

# Publishing EPUB books

EPUB books published with **easybook** are compatible with most e-book readers. The only notable exception are the Kindle readers, that require MOBI format ebooks (as explained in the next chapter).

## 6.1. Templates

The following list shows all the templates used by `epub` format editions to publish the book. These are also the templates that you can override using your own templates: (#custom-templates)

(#custom-templates)

(#custom-templates)

* `code.twig`, `figure.twig`, `chunk.twig`, `layout.twig` and `table.twig` are the same templates and have the same meaning as for the `html_chunked` edition ( (#custom-templates) see `html_chunked` templates). (#html-templates)

  (#html-templates)
* `cover.twig`, the template used to generate the book cover when the book doesn't include its own cover (see the (#html-templates) book cover section bellow). (#epub-cover)
* `content.opf.twig`, the template that generates the `content.opf` file listing all the book contents (including its fonts, images and CSS styles).
* `toc.ncx.twig`, the template that generates the `toc.ncx` file for the book

table of contents.

- `container.xml.twig` y `mimetype.twig`, templates that generate other minor but necessary files for `.epub` format books. Override these templates only if you really know what you're doing, because otherwise, the book won't be properly generated.

(#epub-cover)

## 6.2. Book cover

**easybook** automatically generates a text-based cover for your book, containing both the book title and its author name. If you want to use your very own cover image for the `epub` books, just create a `cover.jpg` file and copy it into one of these directories:

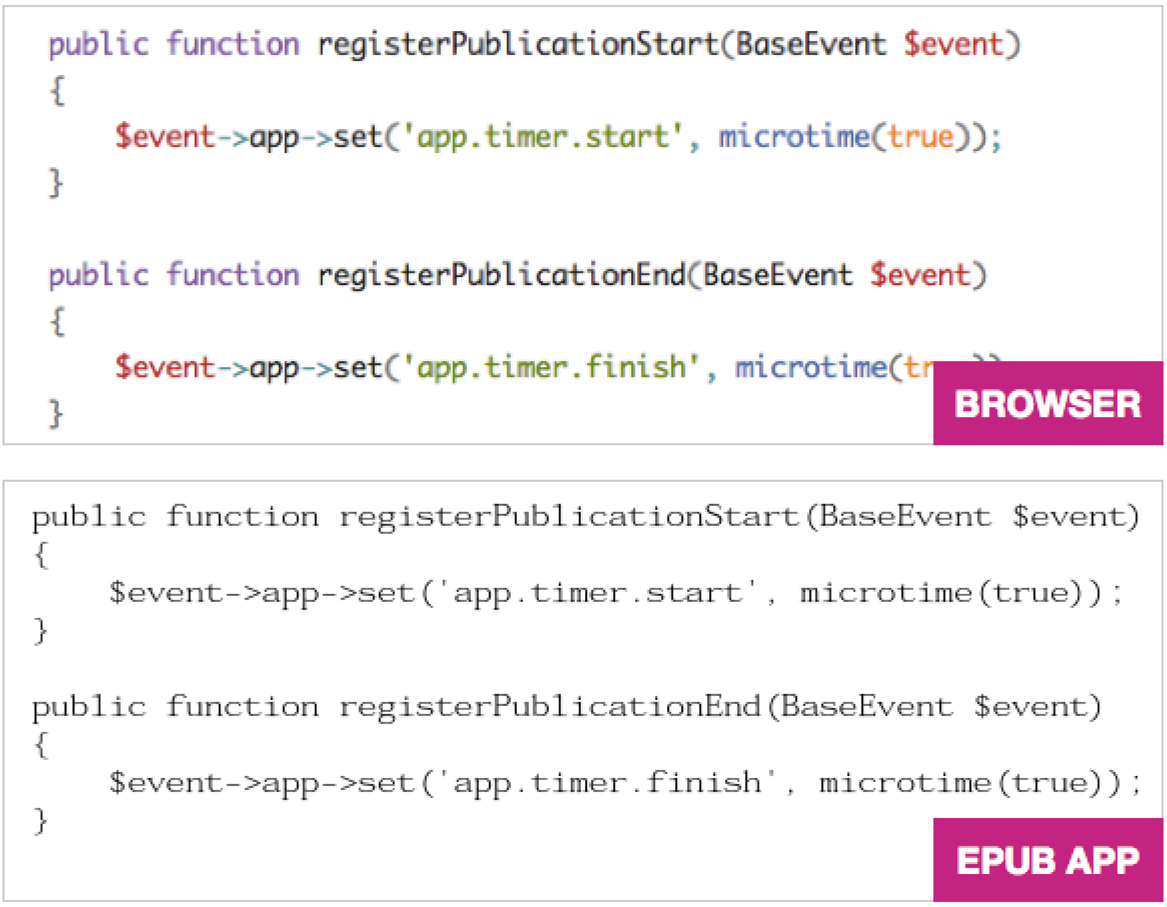
1. `<book>/Resources/Templates/<edition-name>/cover.jpg`, to use the image only for this specific `<edition-name>` edition.
2. `<book>/Resources/Templates/epub/cover.jpg`, to use the same cover image for every `epub` edition.

The cover image format must be JPEG, because this is the most supported format in e-book readers. In order to visualize it correctly in advanced readers such as the iPad, create a large color image (at least 800 pixel height).

## 6.3. Code highlighting

The support of fixed-width fonts in current e-book readers is abysmal. For that reason, syntax highlighting doesn't work as expected. The configuration options are the same as for any other format (`highlight_code` and `highlight_cache`) but the actual result depends completely on the e-book reader used to read the book.

The following screenshot shows a comparison of the same highlighted code displayed in a browser (`html` edition) and in a desktop application to read e-books (`epub` edition):

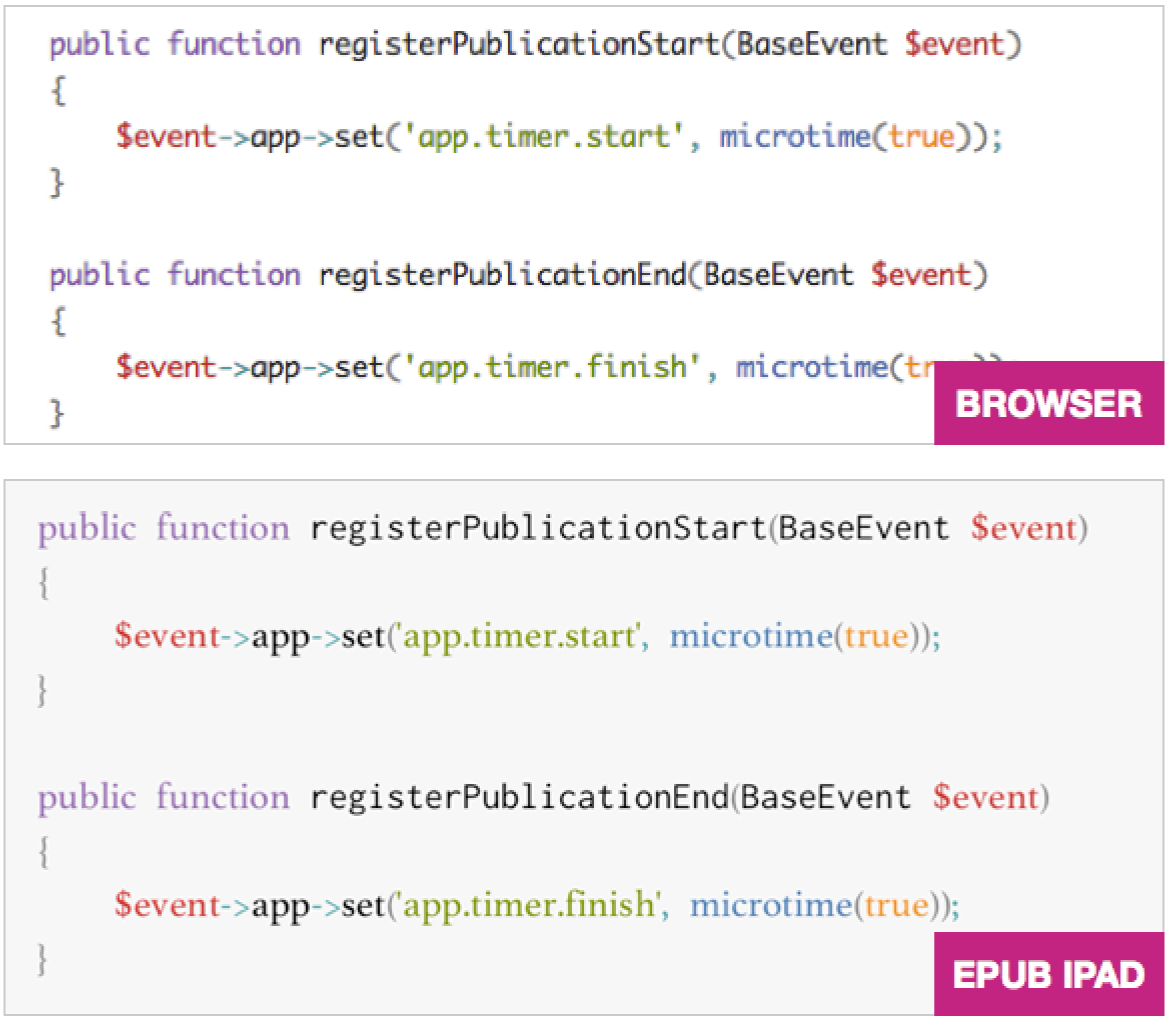


```
public function registerPublicationStart(BaseEvent $event)
{
    $event->app->set('app.timer.start', microtime(true));
}

public function registerPublicationEnd(BaseEvent $event)
{
    $event->app->set('app.timer.finish', microtime(tr
}
```

```
public function registerPublicationStart(BaseEvent $event)
{
    $event->app->set('app.timer.start', microtime(true));
}

public function registerPublicationEnd(BaseEvent $event)
{
    $event->app->set('app.timer.finish', microtime(true));
}
```

**Figure 6.1** Syntax highlighting in the browser vs an epub application

In advanced e-book readers such as the iPad, the code respects the syntax highlighting, but any highlighted code is displayed in the regular book font, instead of the fixed-width font of the non-highlighted code:



```
public function registerPublicationStart(BaseEvent $event)
{
    $event->app->set('app.timer.start', microtime(true));
}

public function registerPublicationEnd(BaseEvent $event)
{
    $event->app->set('app.timer.finish', microtime(tr
}
```

```
public function registerPublicationStart(BaseEvent $event)
{
    $event->app->set('app.timer.start', microtime(true));
}

public function registerPublicationEnd(BaseEvent $event)
{
    $event->app->set('app.timer.finish', microtime(true));
}
```

**Figure 6.2** Syntax highlighting in the browser vs the iPad

---

Considering the actual support for syntax highlighting in e-book readers, it's strongly recommended to disable the code highlighting for epub editions:

---

~~~ .yaml book: # ... editions: ebook: format: epub highlight_code: false # ... ~~~

---

(#epub-cover)
~~~

(#epub-cover)

Chapter 7.

# Publishing MOBI books

MOBI books published with **easybook** are compatible with the Kindle e-book readers.

## 7.1. Requirements

(#epub-cover)

**easybook** uses a free third-party tool called `Kindlegen` to generate MOBI books. For that reason, before publishing any MOBI book, download `Kindlegen` for Windows, Mac OS X or Linux at (#epub-cover) amzn.to/kindlegen (http://amzn.to/kindlegen) .

Once installed, execute the `kindlegen` command without any argument to display the help of the application and to check that everything went fine:

~~~ .cli $ kindlegen

---

Amazon kindlegen V2.9 build 0523-9bd8a95 A command line e-book compiler Copyright Amazon.com and its Affiliates 2013

---

Usage : kindlegen [filename.opf/.htm/.html/.epub/.zip or directory] [-c0 or -c1 or c2] [-verbose] [-western] [-o ] Options: -c0: no compression -c1: standard DOC compression -c2: Kindle huffdic compression

-o : Specifies the output file name. Output file will be created in the same direc-
~~~

tory as that of input file. should not contain directory path.

-verbose: provides more information during ebook conversion

-western: force build of Windows-1252 book

-releasenotes: display release notes

-gif: images are converted to GIF format (no JPEG in the book)

-locale : To display messages in selected language ~~~

### 7.1.1. Configuration

When publishing a `mobi` book, **easybook** looks for the `kindlegen` tool in the most common installation directories depending on the operating system. If `kindlegen` isn't found, **easybook** will ask you to type the absolute path where you installed it:

~~~ .cli $ ./book publish easybook-doc-en kindle

Publishing 'kindle' edition of 'easybook documentation' book...

In order to generate MOBI ebooks, KindleGen library must be installed.

We couldn't find KindleGen executable in any of the following directories: -> /usr/local/bin/kindlegen -> /usr/bin/kindlegen -> c:\KindleGen\kindlegen

If you haven't installed it yet, you can download it freely from Amazon at: http://amzn.to/kindlegen

If you have installed it in a custom directory, please type its full absolute path:

    _ ~~~

In order to avoid typing the `kindlegen` path every time you publish a book, you can leverage the **easybook** global parameters to set the `kindlegen` path once for each book: (#global-parameters)

~~~ .yaml easybook: parameters: kindlegen.path: '/path/to/utils/KindleGen/kindlegen'

---

book: title: '...' # ... ~~~

---

In addition to the `kindlegen` path, you can also tweak the options of the `kindlegen` command with the `kindlegen.command_options` global parameter:

---

~~~ .yaml easybook: parameters: kindlegen.path: '/path/to/utils/KindleGen/kindlegen' kindlegen.command_options: '-c0 -gif verbose'

---

book: title: '...' # ... ~~~

---

## 7.2. Templates

The `mobi` edition format defines the same ten templates of the `epub` format. The reason is that `mobi` books are first generated as `epub` books and then transformed into `mobi` books using the `kindlegen` tool:

---

- `code.twig`, `figure.twig`, `chunk.twig`, `layout.twig` and `table.twig` are the same templates and have the same meaning as for the `html_chunked` edition.
- `cover.twig`, the template used to generate the book cover, which can also use your own image, as explained in the next sections.
- `content.opf.twig`, the template that generates `content.opf` file listing all the book contents.
- `toc.ncx.twig`, the template that generates the `toc.ncx` file for the book table of contents.
- `container.xml.twig` y `mimetype.twig`, templates that generate other minor but necessary files for `.epub` format books.

---

(#global-parameters)
~~~

(#global-parameters)

Chapter 8.

# Publishing PDF books

**easybook** can also publish books as high-quality and full-featured PDF files.

## 8.1. Requirements

(#global-parameters)

**easybook** uses a third-party tool called `PrinceXML` to generate PDF books. For that reason, before publishing any PDF book, download a fully-functional demo version of `PrinceXML` for Windows, Mac OS X or Linux at (#global-parameters) princexml.com (http://www.princexml.com/) .

Once installed, execute the `prince` command without any argument to display the help of the application and to check that everything went fine:

~~~ .cli $ prince Usage: prince [OPTIONS] file.xml Convert file.xml to file.pdf prince [OPTIONS] doc.html -o out.pdf Convert doc.html to out.pdf prince [OPTIONS] FILES... -o out.pdf Combine multiple files to out.pdf

Try 'prince --help' for more information. ~~~

### 8.1.1. Configuration

When publishing a `pdf` book, **easybook** looks for the `PrinceXML` tool in the most common installation directories depending on the operating system. If `PrinceXML` isn't found, **easybook** will ask you to type the absolute path where you installed it:

~~~ .cli $ ./book publish easybook-doc-en print

Publishing 'print' edition of 'easybook documentation' book...

In order to generate PDF files, PrinceXML library must be installed.

We couldn't find PrinceXML executable in any of the following directories: -> /usr/local/bin/prince -> /usr/bin/prince -> C:\Program Files\Prince\engine\bin\prince.exe

If you haven't installed it yet, you can download a fully-functional demo at: http://www.princexml.com/download

If you have installed in a custom directory, please type its full absolute path:

_ ~~~

In order to avoid typing the `PrinceXML` path every time you publish a book, you can leverage the **easybook** global parameters to set the `PrinceXML` path once for each book: (#global-parameters)

~~~ .yaml easybook: parameters: prince.path: '/path/to/utils/PrinceXML/prince'

---

book: title: '...' # ... ~~~

---

## 8.2. Templates

(#global-parameters)

The following list shows all the templates used by `pdf` format editions to publish the book. These are also the templates that you can override (#global-parameters) using your own templates: (#custom-templates)

- A template for each of the 21 content types: `acknowledgement.twig`, `afterword.twig`, `appendix.twig`, `author.twig`, `chapter.twig`, `conclusion.twig`, `cover.twig`, `dedication.twig`, `edition.twig`, `epilogue.twig`, `foreword.twig`, `glossary.twig`, `introduction.twig`, `license.twig`, `lof` (list of figures), `lot` (list of tables), `part.twig`, `preface.twig`, `prologue.twig`, `title.twig` and `toc.twig` (table of contents).
- `book.twig`, this is the final template that assembles all the book contents.
- `code.twig`, the template used to decorate all the code listings included in
~~~

the book.

- `figure.twig`, the template used to decorate all the images included in the book.
- `table.twig`, the template used to decorate all the tables included in the book.

## 8.3. Configuration options

(#custom-templates)

Besides the (#custom-templates) common configuration options, `pdf` editions can also set any of the following specific options: (#common-edition-options)

(#common-edition-options)

- `isbn`, the ISBN-10 or ISBN-13 code of the book.
- `margin`, sets the four margins of the printed book: `top`, `bottom`, `inner` and `outter`. If the book is one-sided, `inner` equals left margin and `outter` equals right margin. The values of the margins can be set with any CSS valid length unit (`1in`, `25mm`, `2.5cm`).

  (#common-edition-options)
- `page_size`, the page size of the printed book. Instead of setting the page dimensions, **easybook** uses (#common-edition-options) named page sizes (http://www.princexml.com/doc/9.0/page-size/) such as `US-Letter`, `US-Legal`, `crown-quarto`, `A4`, `A3`, etc.
- `two_sided`, if `true` the PDF file is formatted for two-sided printing, which means that some blank pages will probably be inserted to ensure that the important contents always start at an odd page. If this option is set to `false`, no blank pages will be inserted.

## 8.4. Book cover

Similarly to ePub books, **easybook** allows PDF books to override the automatically generated text-based cover. If you want to use your very own cover image for the `pdf` books, just create a `cover.pdf` file and copy it into one of these directories: (they are listed by priority, meaning that the first `cover.pdf` file found is used):

1. `<book>/Resources/Templates/<edition-name>/cover.pdf`, to

use the cover only for this specific `<edition-name>` edition.

2. `<book>/Resources/Templates/pdf/cover.pdf`, to use the same cover for every `pdf` edition.
3. `<book>/Resources/Templates/cover.pdf`, produces the same result as the previous option.

The first option is useful when you need to use different covers for different PDF editions of your book (e.g. high quality cover for printing the book, medium quality cover for distributing the book via web, etc.)

Chapter 9.

# Plugins

**easybook** is a rather flexible application with lots of configuration options. However, sometimes your requirements are so specific that cannot be covered by any configuration option.

In those cases, you can use **plugins**, which allow you to completely modify the behavior of **easybook**. During the publication of a book, **easybook** performs a lot of different tasks: loading the book contents, transforming them from Markdown to HTML, building the PDF, EPUB or MOBI file, etc.

Whenever **easybook** begins or ends an important tasks, an **event** is notified. Plugins can *hook* into any of these events to perform any action before or after these tasks.

## 9.1. Creating the first plugin

**easybook** uses the Event Dispatcher component (http://symfony.com/doc/current/components/event_dispatcher) of Symfony2 to define its plugins. Technically speaking, a plugin is any PHP class stored in the `Resources/Plugins/` directory of your book, whose name ends with the word `Plugin` and that implements the `EventSubscriberInterface` interface.

Imagine that you want to measure the time elapsed to publish a book. Given that **easybook** notifies both the begin and the end of the book publication, it's really easy to create a plugin that *hooks* on both events to register the timestamp of each moment.

If the plugin is called `Timer`, create a new `TimerPlugin` class in the `<book>/Resources/Plugins/TimerPlugin.php` file and add the following content:

~~~ .php // /Resources/Plugins/TimerPlugin.php use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class TimerPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array(); } } ~~~

Now, use the `getSubscribedEvents()` method to define the specific events that you need to *hook* into. In the next section, you'll find the complete list of easybook events but for now, just assume that the event notified when the book publication begins is called `Events::PRE_PUBLISH` and the other event is called `Events::POST_PUBLISH`: (#easybook-events)

~~~ .php // /Resources/Plugins/TimerPlugin.php use Symfony\Component\EventDispatcher\EventSubscriberInterface; use Easybook\Events\EasybookEvents as Events; use Easybook\Events\BaseEvent;

---

class TimerPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( Events::PRE_PUBLISH => 'registerPublicationStart', Events::POST_PUBLISH => 'registerPublicationEnd', ); }

```
public function registerPublicationStart(BaseEvent $even
t)
{
    // ...
}

public function registerPublicationEnd(BaseEvent $event)
{
    // ...
}
```

---

} ~~~

---

First, the `getSubscribedEvents()` method returns an array of events that you want to subscribe to and the name of the methods that will be executed when those events occur. Therefore, this plugin must define two new methods called `registerPublicationStart()` and `registerPublicationEnd()`.

(#easybook-events)

The first argument of each method (`BaseEvent $event`) is automatically injected by **easybook** and it allows to access easily to any application property or method. In the next section, you'll discover all the properties of the (#easybook-events) BaseEvent class. (#baseevent-class)

The class of the argument injected to the plugin methods depend upon the kind of event notified. In the above code, the object is an instance of `BaseEvent`, but for other events, the object is an instance of the `ParseEvent` class.

---

Finally, to register the begin and the end of the book publication, you can simply store the timestamp of each moment in some application properties:

---

~~~ .php // /Resources/Plugins/TimerPlugin.php use Symfony\Component\EventDispatcher\EventSubscriberInterface; use Easybook\Events\EasybookEvents as Events; use Easybook\Events\BaseEvent;

---

class TimerPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( Events::PRE_PUBLISH => 'registerPublicationStart', Events::POST_PUBLISH => 'registerPublicationEnd', ); }

```
public function registerPublicationStart(BaseEvent $even
t)
{
    $event->app->set('app.timer.start', microtime(tru
e));
}

public function registerPublicationEnd(BaseEvent $event)
{
    $event->app->set('app.timer.finish', microtime(tru
e));
}
```

---
~~~

} ~~~

## 9.2. Events defined by easybook

(#baseevent-class)

In order to keep things organized and to avoid the problems introduced by the (#baseevent-class) magic strings (http://en.wikipedia.org/wiki/Magic_string) , **easybook** follows the best practice of defining the event names in a special class:

~~~ .php namespace Easybook\Events;

final class EasybookEvents { const PRE_NEW = 'book.new.start'; const POST_NEW = 'book.new.finish'; const PRE_PUBLISH = 'book.publish.start'; const POST_PUBLISH = 'book.publish.finish'; const PRE_PARSE = 'item.parse.start'; const POST_PARSE = 'item.parse.finish'; const PRE_DECORATE = 'item.decorate.start'; const POST_DECORATE = 'item.decorate.finish'; } ~~~

The following table explains when is each event notified and the kind of event object injected to the subscriber methods:

| Event | Injected Object | Notification |
|---|---|---|
| `PRE_NEW` | `BaseEvent` | When creating a new book, after validating its title and before creating the file and directory structure |
| `POST_NEW` | `BaseEvent` | When creating a new book, after creating its file and directory structure and before displaying the success message to the user |
| `PRE_PUBLISH` | `BaseEvent` | When publishing a book, after all its configuration has been loaded and validated and after executing the `before_publish` scripts |
| `POST_PUBLISH` | `BaseEvent` | When publishing a book, after the publication is completed and before executing the `after_publish` scripts |
| `PRE_PARSE` | `ParseEvent` | When publishing a book, just before each |
~~~

| Event | Injected Object | Notification |
|---|---|---|
| | | book content is transformed from Markdown into HTML |
| `POST_PARSE` | `ParseEvent` | When publishing a book, just after each book content is transformed from Markdown into HTML |
| `PRE_DECORATE` | `BaseEvent` | When publishing a book, just before each book content is decorated with the appropriate Twig template |
| `POST_DECORATE` | `BaseEvent` | When publishing a book, just after each book content is decorated with the appropriate Twig template |

### 9.2.1. The BaseEvent class

This is the generic event class that it's used by most of the event notifications. It only defines one property:

* `app`, it stores the object that represents the whole **easybook** application. Access to any application property with `$event->app['property_name']` and modify any property with `$event->app['property_name'] = 'property_value';`

In addition, it defines two methods:

* `getItem()`, it returns the active book item. When used for example with the `PRE_DECORATE` event, it stores the book item that is about to be decorated with the Twig template.
* `setItem(array $item)`, it allows you to replace the active item by the given item. This way you can modify any item configuration or event the whole item content.

### 9.2.2. The ParseEvent class

This class extends the previous `BaseEvent` class and adds new methods to ease the access to the properties of the item being parsed. The only property defined by this class is the same `app` property explained in the previous section.

This class defines four methods:

- `getItem()`, it returns the active item that it's being parsed.
- `setItem(array $item)`, it allows you to replace the active item being parsed with your own item.
- `getItemProperty($key)`, it returns the value of the given `$key` property of the item. To get for example the original Markdown content of the item being parsed, use `$event->getItemProperty('original')`
- `setItemProperty($key, $value)`, it modifies the value of the `$key` property of the item with the `$value` content. To replace for example the original Markdown content of the item being parsed, use `$event->setItemProperty('original', '...')`

## 9.3. Creating an advanced plugin

### 9.3.1. Subscribing to several events

In order to write more legible code, you may want to split the plugin operations into several methods. When subscribing to the event, instead of passing a simple string with the method name, pass an array with all the method names:

~~~ .php class MyPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( // three different methods subscribed to the same event Events::PRE_PUBLISH => array( 'registerPublicationStart', 'fixPublicationDirectory', 'prepareBookContents', ) ); } ~~~

### 9.3.2. Event priorities

When several methods are subscribed to the same event, the order in which they are executed is determined according to these rules:

1. **easybook** loads the plugin classes alphabetically, so the `AaaPlugin` methods are executed before the methods of the `BbbPlugin` class.
2. Inside a single class, the methods are executed in the same order as they were registered.

However, it's really easy to modify this behavior by setting explicitly the priority of each subscribed method. When subscribing to the event, instead of passing a simple string with the method name, pass an array with the method name and its priority:

~~~ .php class MyPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( Events::PRE_PUBLISH =>

array('firstMethodName', 10), Events::POST_PUBLISH => array('secondMethodName', -200), ); } } ~~~

The priority of each method is defined as an integer (positive or negative). The default priority is `0` and higher values mean more priority. You can also set priority when subscribing several methods to the same event:

~~~ .php class MyPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( // three different methods subscribed to the same event Events::PRE_PUBLISH => array( array('firstMethodName', 10), array('secondMethodName', 20), array('thirdMethodName', 30), ) ); } } ~~~

### 9.3.3. Restricting the plugin execution

Sometimes the execution of some plugin methods only make sense for a particular edition or for a specific edition format (e.g. only for PDF books). The easiest way to restrict the execution of the plugin is to get the edition being published and check its name or format:

~~~ .php use Symfony\Component\EventDispatcher\EventSubscriberInterface; use Easybook\Events\EasybookEvents as Events; use Easybook\Events\BaseEvent;

class MyPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( Events::PRE_PUBLISH => 'myMethod', ); }

```
public function myMethod(BaseEvent $event)
{
    if ('pdf' == $event->app->edition('format')) {
        // this code is only executed for PDF books
    }

    if ('my_edition' == $event->app['publishing.editio
n']) {
        // this code is only executed when publishing th
e
        // 'my_edition' edition of the book
    }
}
```

} ~~~

You can also stop the propagation of the event to prevent other plugins' methods from executing. To do so, invoke the `stopPropagation()` method of the event object:

~~~ .php use Symfony\Component\EventDispatcher\EventSubscriberInterface; use Easybook\Events\EasybookEvents as Events; use Easybook\Events\BaseEvent;

class MyPlugin implements EventSubscriberInterface { public static function getSubscribedEvents() { return array( Events::PRE_PUBLISH => 'myMethod', ); }

```
public function myMethod(BaseEvent $event)
{
    // do something ...

    // stop the event propagation (no other method or
    // plugin will be executed to respond to this event)
    $event->stopPropagation();
}
```

} ~~~

## 9.4. Built-in plugins

**easybook** relies heavily on plugins to perform some of the most important tasks when publishing a book. The source code of these plugins is a good resource to learn how to develop advanced plugins:

- `CodePlugin`, it is used for syntax highlighting the code blocks.
- `ImagePlugin`, it decorates each image with its own Twig template and label. It also fixes the URL of the images when the books is published as a website and it generates the `lof` (*list of figures*) special content.
- `LinkPlugin`, it fixes the internal links of the books published as websites.
- `ParserPlugin`, it adds labels to the section headings and it performs some minor fixes in the HTML generated by the Markdown transformer.
- `TablePlugin`, it decorates each table with its own Twig template and label. It also generates the `lot` (*list of tables*) special content.
- `TimerPlugin`, it's used to measure the time elapsed to publish a book.
~~~

Chapter 10.

# Hacking easybook

## 10.1. Custom configuration options

**easybook** doesn't restrict the configuration options that you can set on your book, editions and contents. **easybook** always puts you in control, so you can define all the new settings that you may need.

Do you want to show the price of the book on the cover? Add a new option called `price` under the `book` option of `config.yml` file:

~~~ .yaml book: # ... price: 10 ~~~

Now you can use this option in any template with the following expression: `{{ book.price }}`.

Do you want to use different CSS frameworks to generate the book website? Add a new option called `framework` in some editions:

~~~ .yaml editions: my_website1: format: html_chunked framework: bootstrap_3 # ...

```
my_website2:
    extends:   my_website1
    framework: foundation_4
```

~~~

This new option is now available in any template through the following
~~~

expression: `{{ edition.framework }}`.

You can also add new options to the contents of the book. Do you want to indicate the estimated reading time in each chapter? Add the following `time` option for `chapter` elements:

~~~ .yaml contents: - { element: cover } - ... - { element: chapter, number: 1, ..., time: 20 } - { element: chapter, number: 2, ..., time: 12 } - ... ~~~

And now you can add `{{ item.config.time }}` in `chapter.twig` template to show the estimated reading time for each chapter.

Book configuration options can also use Twig expressions for their values (even for dynamic options set by `--configuration` option explained in the next section):

~~~ .yaml book: title: "{{ book.author }} diary" author: "..." publication_date: "{{ '+2days'|date('m/d/Y') }}" # ... ~~~

## 10.2. Dynamic configuration options

Sometimes, configuration option values can't be defined or are variables themseleves. For that reason, **easybook** allows you to set or override configuration options dynamically with the `publish` command.

Add the `--configuration` option to `publish` command and pass to it a JSON formatted string with the configuration options:

~~~ .cli $ ./book publish the-origin-of-species web --configuration='{ "book": { "title": "My new title" } }' ~~~

Using dynamic options you can for example define a new `buyer` option that stores the name of the person that bought the book. Then you can prevent or minimize digital piracy displaying the buyer name inside the book (use `{{ book.buyer }}` Twig expression in any book template):

~~~ .cli $ ./book publish the-origin-of-species print --configuration='{ "book": { "buyer": "John Smith" } }' ~~~

Likewise, any edition option can be set dynamically:

~~~ .cli $ ./book publish the-origin-of-species web --configuration='{ "book": { "editions": { "web": { "highlight_code": true } } } }' ~~~
~~~

When passing a lot of configuration options, you must be very careful with JSON braces and quotes. If you run the command automatically, it is easier to create a PHP array with all the options and then convert it to JSON with the `json_encode()` function.

Dynamic options are so advanced that they even allow you to define on-the-fly editions:

~~~ .cli $ ./book publish the-origin-of-species edition1 --configuration='{ "book": { "editions": { "edition1": { ... } } } }' ~~~

When publishing a book, **easybook** uses the following priority hierarchy to combine configuration options (the higher, the more priority):

1. Dynamic options set with `--configuration` option.
2. Options set in the `config.yml` file.
3. Default options defined by **easybook**.

Configuration options related to editions also take into account the possible edition inheritance. If an edition inherits from another one, its options always override the options of its *parent* edition.

## 10.3. Configuring global parameters

In addition to its own configuration options, a book can also modify global **easybook** parameters. To do so, add an `easybook` configuration section inside your `config.yml` file (it's recommended to create it at the top of the file to spot it easily):

~~~ .yaml easybook: parameters: kindlegen.command_options: '-c0 -gif verbose' kindlegen.path: '/path/to/utils/KindleGen/kindlegen'

book: title: '...' # ... ~~~

Define the **easybook** global options under the `parameters` key of `easybook` section. In the previous example, the book sets the `kindlegen.command_options` option to tweak the command used to generate Kindle-compatible MOBI files. In addition, to avoid repeating it each time the book is generated, the book also sets the path to the `kindlegen` library thanks to the `kindlegen.path`.

You can replace any of the parameters defined or used in the `Application.php` class located at `src/Easybook/DependencyInjection/`. Therefore, use the following configuration to modify the directory where the book is generated:

~~~ .yaml easybook: parameters: publishing.dir.output: '/my/path/for/books/my-book'

book: title: '...' # ... ~~~

The separator used by the slugger is another option that is usually modified. By default **easybook** uses the dash (`-`) for the slugs (this affects the book page names and the URL for the books published as websites). If you prefer the underscore (_) you can now easily configure it in your book:

~~~ .yaml easybook: parameters: slugger.options: separator: '_'

book: title: '...' # ... ~~~

## 10.4. Defining new content types

It's uncommon and generally unneeded, but you can also define new content types besides the 21 types included in **easybook**. Imagine that you want to include a page with a humorous cartoon between some chapters. Let's call this new content type `cartoon`.

If pages of type `cartoon` include few contents (a picture and its caption for example), it's better to define these contents directly in the `config.yml` file:

~~~ .yaml contents: - { element: cover } - ... - { element: chapter, number: 1, ... } - { element: cartoon, image: cartoon1.png, caption: '...' } - { element: chapter, number: 2, ... } - ... ~~~

Then, create a custom `cartoon.twig` template in the `Resources/Templates/` directory of your book:

~~~ .twig

{{ item.config.caption }}

~~~

In contrast, if pages of type `cartoon` include a lot of contents, it's better to create a content file for each of these elements:
~~~

~~~ .yaml contents: - { element: cover } - ... - { element: chapter, number: 1, ... } - { element: cartoon, content: cartoon1.md } - { element: chapter, number: 2, ... } - ... ~~~

Then, display this content with the following simplified `cartoon.twig` template:

~~~ .twig

{{ item.content }}
~~~

Finally, you can also combine these two methods creating a file with the contents and adding several configuration options in `config.yml`. The template can easily use both sources of information:

~~~ .twig

```
{{ item.content }}
```

~~~

That's it! You can now use the new `cartoon` content type in your book and you can create new content types following the same steps explained above.

## 10.5. easybook internals

**easybook** flexibility allows you to create advanced books without effort and without having to study the source code of the application. However, to master **easybook** you'll have to dive into the guts of the application.

The most important class of **easybook** is `src/Easybook/DependencyInjection/Application.php`. This class follows the *dependency injection* pattern, uses the Pimple component (http://pimple.sensiolabs.org/) and contains all the variables, functions and services of the application.

The most interesting command of **easybook** is `publish`, which publishes an specific edition of the book. Internally it uses a `*Publisher` class which depends on the type of edition that is published (`epub`, `mobi`, `pdf`, `html` or `html_chunked`). The details of each *publisher* vary, but the basic workflow is always the same:
~~~

~~~ .php public function publishBook() { $this->loadContents(); $this->parseContents(); $this->decorateContents(); $this->assembleBook(); } ~~~

First, the contents of the book defined in the `contents` option of the `config.yml` file are loaded (`loadContents()`) . Then, each content is parsed (`parseContents()`) to convert it from its original format to the format needed by this publisher.

At the moment **easybook** only supports Markdown as the original format. If you want to add support for other formats, you have to create a new parser ( dig into the Markdown parser to know how to create it) and you also have to change the `parseContents()` method of the publisher.

After converting all contents to the desired format (usually HTML) they are decorated using Twig templates (`decorateContents()`). Finally, the `assembleBook()` method is responsible for creating the published book. This is the most unique method of the publishers, as sometimes it has to create a PDF file, sometimes just an HTML file and other times it must create an entire website with many HTML pages.
~~~

Appendix A.

# Other First Level Header

**easybook** uses Markdown as the standard markup language to create books and contents. In the future, **easybook** will support other markup languages such as reStructuredText. Meanwhile, **easybook** enhances the original Markdown syntax to provide some of the most demanded features for book writers.

## Basic syntax

**easybook** supports all of the original features described in the official Markdown syntax (http://daringfireball.net/projects/markdown/syntax/) . Please, refer to that reference to learn how to define headers, paragraphs, lists, code blocks, images, and so on.

## Extended syntax

**easybook** includes a Markdown parser based on PHP Markdown Extra Project (http://michelf.com/projects/php-markdown/extra/) . This project brings some nice extra features to the basic Markdown syntax.

### Header id attributes

You can add custom `id` attributes to any header (both *setext* and *atx* style headers):

~~~ (atx style headers)
~~~

# Appendix A. Header 1

...

## Header 2

(setex style headers)

# Appendix A. Other First Level Header

~~~

Then, you can link internally to any book section:

~~~ In the first section you can see the differences between (#header1) this section and (#my-custom-header2-id) that section. ~~~ (#special-id)

As a bonus, in PDF books the internal links display the linked book page.

## Tables

You can add leading and trailing pipes to table rows:

~~~ | First Header | Second Header | | ------------- | ------------- | | Content Cell | Content Cell | | Content Cell | Content Cell | ~~~
~~~

| First Header | Second Header |
|---|---|
| Content Cell | Content Cell |
| Content Cell | Content Cell |

The contents of the cells can be aligned adding a `:` character on the side where you want contents aligned. In the following example, the contents of the first column will be left-aligned and the contents of the second column will be right-aligned:

~~~ | Item | Value | | :-------- | -----:| | Computer | $1600 | | Phone | $12 | | Pipe | $1 | ~~~

Table contents can include any simple formatting as bold, italics, code, etc.

### Other features

(#special-id)

As explained in the (#special-id) official PHP Markdown Extra syntax (http://michelf.com/projects/php-markdown/extra/) you can also create definition lists, footnotes, automatic abbreviations, fenced code blocks, etc.

## easybook exclusive syntax

### Code blocks

Code blocks can be automatically highlighted using the following syntax:

~~~ [code language] ...

[code php] ...

[code xml] ... ~~~

**easybook** recognizes automatically tens of programming languages thanks to the use of GeSHi highlighting library (http://qbnz.com/highlighter/) .

### Image alignment

(Added in easybook 4.8)

Books usually align/float images on the right/left of the contents, but Markdown doesn't include a mechanism to define image alignment:

~~~

Alt text
**Figure A.1** Alt text

~~~

**easybook** defines a simple and Markdown-compatible mechanism based on adding whitespaces on *Alt text*:

~~~ // regular image not aligned

Test image
**Figure A.2** Test image

// "alt text" has a whitespace on the left // -> the image is left aligned

Test image
**Figure A.3** Test image

// "alt text" has a whitespace on the right // -> the image is right aligned

Test image
**Figure A.4** Test image

// "alt text" has whitespaces both on the left and on the right // -> the image is centered

Test image
**Figure A.5** Test image

~~~

If you enclose alt text with quotes, make sure that whitespaces are placed outside the quotes. The following images for example don't define any alignment:

~~~

"Test image"
**Figure A.6** "Test image"

" Test image"
**Figure A.7** " Test image"
~~~

"Test image "
**Figure A.8** "Test image "

" Test image "
**Figure A.9** " Test image "

~~~

Image alignment is also possible when using the alternative image syntax:

~~~

Test image
**Figure A.10** Test image

Test image image **Figure A.11** Test image

Test image
**Figure A.13** Test image

~~~

Test image **Figure A.12** Test image

## Decorative images

(Added in easybook 5.0)

**easybook** makes really easy defining images that will be treated as book illustrations (*figures*) and automatically labeled and numbered. The `label` configuration option defines whether all the images in the book are to be treated as illustrations, but it has one drawback: it applies to all of the images, so when set to `true` you cannot include other purely decorative images (for example, a graphical separator for paragraphs or sections).

The following syntax makes possible defining this type of decorative images:

~~~ ~~~

That is: just defining '*' as the image title tells **easybook** that the image is not to be treated as an illustration but a normal image (i.e. it will not have caption nor will be included in the table of figures, and it will be embedded into the normal text flow instead of being assigned a block format).

All other image options also work, so the following examples are still valid:

~~~
~~~

~~~

## Page breaks

(Added in easybook 5.0)

Books can force page breaks inside any content by using one of the following two special tags:

~~~

~~~

The first tag uses the LeanPub syntax and the second one uses the Marked syntax. Note that you must write the tags as shown above, without adding any extra whitespace.

You can mix the two tags in the same content and you can place them anywhere (inside a table, inside a list, inside a heading, etc.)

## Admonitions

(Added in easybook 5.0)

**easybook** supports several kinds of admonitions. The syntax is based on LeanPub and Marked and it's very similar to blockquotes:

~~~

> This is a regular bloquote Nothing special here

This is a sidebar or aside You can use **any markup** inside

A>

> even nested blockquotes

A>

And lists:

A>

- Item 1
- Item 2
- Item 3

This is a note ...

This is a tip ...

This is an error message ...

This is just an information ...

This is a question ...

This is a discussion ...

∼∼∼