MANUAL

LAIKA

Table of Content

ABOUT LAIKA

RUNNING LAIKA

PREPARING CONTENT

CUSTOMIZING LAIKA

EXTENDING LAIKA

SUB-MODULES

REFERENCE

Migration Guide 199

Features

Laika is a customizable and extensible toolkit for transforming lightweight text markup and template based site and e-book generation.

It can be used in a range of usage scenarios:

- As an sbt plugin as part of your build and CI pipeline, e.g. for generating Scala project documentation

- As part of your build with other build tools such as Mill through its library API.

- In server-side Scala applications, e.g. for transforming user input in markup syntax or as part of document processing pipelines

- In browser-based applications via its library API which is published for Scala.js

- Scaling up to producing complex e-books and sites from dozens of inputs, to scaling down to transform a snippet of Markdown with three lines of code.

Supported Platforms

Laika can be used as:

- A plugin for sbt version 1.x.

- A library on the JVM with Scala 2.12 or 2.13.

- A library in Scala.js 1.0 applications with all its features except File/Stream IO, EPUB and PDF output

For getting started see the documentation of the **sbt Plugin** or **Library API**.

Supported Formats

- Markdown including GitHub Flavor

- reStructuredText including its standard directives

- HOCON for configuration and directive attributes, based on Laika's own parser

- HTML site generation based on templates

- **E-Books (EPUB & PDF)** with auto-generated navigation

- Lightweight default theme with configurable styling for site, EPUB and PDF output with extensive **Theme Settings**.

Content Organization

- **Validated Internal Links** in "native" text markup syntax, based on relative paths or **Linking by Section Headline**.

- Avoid repetition and centralize all external URLs used in links with **Global Link Definitions**.

- Use Laika's **Standard Directives** for automatically **Generating Navigation Trees** or **Breadcrumbs**, optionally with **Auto-Numbering**.

- Use integrated parsers for **Syntax Highlighting** that work for all output formats. Supported out of the box are Scala, Dotty, Java, Python, JavaScript (JSX), TypeScript (TSX), HTML, CSS, XML, JSON, HOCON, SQL, EBNF

- Freely organize and merge content from multiple input directories or generated in-memory with the library's **Virtual Tree Abstraction**.

Library API

- Use a purely functional API, with full referential transparency, no exceptions and no runtime reflection.

- Abstracts all effectful computations behind a polymorphic effect type based on `cats-effect` typeclasses, so that any compatible library can be used with Laika (`cats.IO`, `Monix`, `Zio`).

- Full control over ExecutionContexts for blocking IO and CPU-bound processing.

Customizations

- Create your own look & feel by **Creating Themes** or adjust the built-in theme by **Creating Templates**.

- Process and transform **The Document AST** between parsing and rendering by **AST Rewriting**.

- Adjust the rendered output for individual AST node types by **Overriding Renderers**.

Extensibility

- Extend the syntax of text markup languages, either by **Implementing Directives** or by **Writing Parser Extensions**.

- Support additional languages for Laika's built-in highlighter by **Adding Syntax Highlighters**.

- Use simple contracts and APIs for adding **New Markup or Output Formats**.

- Write all parsers based on **Laika's Parser Combinators** which are optimized for multi-pass markup parsing.

Design Goals

Laika is designed around a few guiding principles that are outlined below.

They may be of interest for finding out how Laika tries to differentiate itself from comparable tools used in the Scala ecosystem. It may also be relevant for potential contributors as a very high-level introduction to the library's underlying design choices.

Not Tied to External Tools

Laika is released as a standalone library. It does not rely on any external tool or language installed on your machine, apart from the JVM or the browser, depending on whether you use it with Scala.js or not.

The Scala ecosystem still heavily relies on installed tools for project documentation. While it is attractive to benefit from the wide adoption of projects like Jekyll or Hugo and all its available themes and extensions, the fact that a contributor has to perform installations on her machine just to verify a small documentation change presents an extra hurdle that would be nice to avoid.

Scala project documentation is not the only use case Laika supports though. Its independence from tools also makes it easier to embed it in server-side or browser applications.

Not Tied to the JVM

The `laika-core` module, about 80% - 90% of the library's functionality is published for both, the JVM and Scala.js. This way Laika can be used in the browser, e.g. for processing user input right on the client.

The implementation is also free of any runtime reflection, a prerequisite for Scala.js support and has minimal dependencies. `laika-core` only depends on `cats-core` while the `laika-io` module adds a dependency on `cats-effect`.

The only module definitely tied to the JVM is the `laika-pdf` module which depends on a fairly heavy Java library (Apache FOP).

Not Tied to sbt

While Laika does include an sbt plugin, it is a separate artifact and only a very thin layer of sbt settings and tasks on top of the library API. In fact, Laika started as a library and only added the sbt plugin in version 0.5 in 2014.

Due to this design, less than 2% of the code lines in the Laika project depend on the sbt APIs. Laika can easily be used with other build tools such as Mill or as a library in application code.

This is in contrast to many sbt plugins which "bury" the provided functionality under the sbt APIs, making it unavailable for any other use case.

Not Tied to Markdown

Even though Markdown has become fairly ubiquitous, Laika avoided tying its internal design entirely to this one markup format, in contrast to many similar tools and libraries. It is decoupled from Markdown in several ways:

- In addition to its Markdown parser it comes with support for a second, advanced text markup format right out of the box: reStructuredText. Which brings advanced table formats and footnotes to the table. It is probably the only complete and actively maintained implementation of reStructuredText on the JVM.

- The Markdown parser produces a generic AST model as a result, instead of directly producing an output format. This AST does not reflect any specific semantics of Markdown and other text markup formats produce the exact same kind of model. The AST can get analyzed and transformed before being fed to any of the supported renderers. (See **The Document AST** and **AST Rewriting** for details)

- The Markdown and reStructuredText parser sit behind a well-defined (and fairly simple) abstraction, the `MarkupFormat` trait. This allows 3rd-parties (or future Laika releases) to add support for additional markup formats while still benefiting from the subsequent processing steps the library provides, like validating links or producing tables of contents. (See **New Markup or Output Formats** for details)

- Finally there is another hook for extending the syntax of existing text markup parsers. A syntax extension can be written in such a way that it works with any markup language as long as the syntax does not conflict with its "native" syntax. (See **Writing Parser Extensions** for details)

Not Tied to HTML

Similar to Laika's independence from a particular input format, it also avoids the coupling to HTML output found in many other tools.

It supports HTML, EPUB and PDF out of the box, together with an `AST` renderer that provides a formatted representation of the document AST for debugging purposes.

All renderers will be fed with an AST model which is usually obtained by parsing text markup, but can alternatively be generated in memory (or a combination of the two). (See **The Document AST** for details)

Like the parsers the renderers are all based on a well-defined abstraction, the `RenderFormat` trait. This allows 3rd-parties (or future Laika releases) to add support for additional output formats. (See **New Markup or Output Formats** for details)

Not Tied to the File System

While the most common forms of specifying the input and output are file-based with the default settings of the sbt plugin specifying a single directory as input, this is by no means the only option to compose the sources.

Laika internally uses a **Virtual Tree Abstraction** to organize markup documents, templates and configuration files. The decoupling from the file system gives additional flexibility.

For example, when two directories are specified as input the resulting, logical tree will be a recursive merge of the two trees. This way reusable templates, styles and static files can be kept separate from the markup files.

A second example is the need to add generated content to the files from the input directory. The content can simply be generated in-memory and get a virtual path assigned that will cause the generated content to be "mounted" at the specified location. There is no need to write the content back to the file system first, which is also very CI-friendly.

All internal link references between documents happen based on the virtual path, meaning a path in the form of `../intro.md` inside a file may refer to something generated in-memory.

Purely Functional

All Laika APIs are designed to be purely functional. The only minor exception is the implementation of the sbt plugin (< 2% of Laika's code), which has to code against sbt's impure APIs.

The library API is fully referentially transparent and abstracts all effectful computations behind a polymorphic effect type based on `cats-effect` typeclasses, so that any compatible library can be used with Laika (`cats.IO`, `Monix`, `Zio`).

While the target audience of this project is larger than the purely functional part of the Scala community, it is always easier to ignore the aspect of referential transparency of a library that supports it in case it is not your design approach than the other way round: trying to wrap pure code around an impure library.

A Toolkit for Creating Toolkits

This design goal is mostly achieved "for free" by following the previously mentioned goals of decoupling on many levels, but still deserves an explicit entry.

While Laika does also aim to be a useful tool right out of the box, the fact that almost every parsing, processing or rendering step can either be heavily customized or replaced in its entirety by alternative implementations, makes it a good candidate for building other engines on top of it which offer a different focus of use cases.

sbt Plugin

The core of the Laika toolkit does not depend on sbt in any way. The sbt plugin merely provides a thin layer on top of the library API, exposing Laika's feature set as sbt tasks and settings.

This chapter only covers the topics which are specific to using Laika with sbt, while most other parts of the manual apply to both use cases, library and plugin.

Adding the Plugin to the Build

Laika's sbt plugin requires sbt 1.x.

If you still need to use sbt 0.13.x, you can use Laika 0.7.0 which was the last release supporting that sbt version.

First add the plugin to `project/plugins.sbt`:

```
addSbtPlugin("org.planet42" % "laika-sbt" % "0.16.1")
```

Then enable the plugin in your project's `build.sbt`:

```
enablePlugins(LaikaPlugin)
```

Preparing Content

The plugin expects the source files in the `src/docs` directory inside your project. This default can be overridden with:

```
Laika / sourceDirectories := Seq(target.value / "somewhere-else")
```

You can provide multiple input directories which will be merged into Laika's **Virtual Tree Abstraction**.

Laika does not have any special directories and content can be nested in sub-directories down to arbitrary levels. For more details on how to organize content, see **Directory Structure**.

The plugin distinguishes between the following file types:

- **Markup Files**: Files with the extensions `.md`, `.markdown` or `.rst` will be parsed and rendered to the target in the same directory structure and with the same file names apart from the suffix, which will be replaced depending on the output format (e.g. `.html`).

- **Configuration Files**: Each directory can contain an optional `directory.conf` file for specifying things like navigation order or chapter title. See **Configuration Files** for details.

- **Template Files**: You can provide a default template per directory with the name `default.template.<suffix>` with the suffix matching the output format (e.g. `.html`). They will also be

applied to sub-directories, unless overridden. Default templates in one of your input directories always override default templates provided by themes. You can also add additional templates with the name pattern `*.template.<suffix>`, which will only be applied when a markup document explicitly refers to them in its configuration header.

- **Static Files**: All other files, like CSS, JavaScript, images, etc., will be copied over to the target in the same directory structure and with identical file names.

Generating a Site

For parsing your markup files and generating a site simply run:

```
laikaSite
```

This will parse all files with the extensions `.md`, `.markdown` or `.rst` and render them as `.html` files in the same directory structure as the input.

It will also copy all other file types over to the target folder (apart from templates and laika configuration files).

The site will be generated in the directory `target/docs/site` within your project. The default can be changed with:

```
laikaSite / target := target.value / "somewhere-else"
```

Laika builds on sbt's caching features, therefore if neither any input files have been modified nor any output files removed, it is a no-op.

Including EPUB and PDF

EPUB and/or PDF files can be included with these settings:

```
laikaIncludeEPUB := true
laikaIncludePDF  := true
```

In this case the `laikaSite` task will generate EPUB and/or PDF files in the root directory of your output, in addition to the generated HTML site. The name of the files will be `<project-name>-<version>.<epub|pdf>` by default. You can change it with:

```
laikaEPUB / artifactPath  := target.value / "my-docs.epub"
laikaPDF / artifactPath   := target.value / "my-docs.pdf"
```

For other e-book configuration options like navigation structures and cover images see **E-Books (EPUB & PDF)**.

Including Scaladoc

The following setting causes the scaladoc output of your project to be included in the generated site:

```
laikaIncludeAPI := true
```

It will be copied into the `api` directory of your site's target, unless you change its setting:

```
laikaCopyAPI / target := (laikaSite / target).value / "somewhere-else"
```

Generating Individual Formats

Instead of using the `laikaSite` task, you can also selectively pick the output formats to generate on each run with the `laikaGenerate` task:

```
laikaGenerate html epub pdf
```

This will parse all files with the extensions `.md`, `.markdown` or `.rst` and render them in the specified list of formats. It is efficient in that the parsing of markup files only happens once to obtain a document AST and use this structure to render all formats.

Valid format arguments are `html`, `epub`, `pdf`, `xslfo`, `ast`. The latter is a debug output that renders the document AST in a formatted structure.

Finally if you only work with a single format there are also shortcut tasks for those: `laikaHTML`, `laikaEPUB`, `laikaPDF` and `laikaAST`.

Plugin Settings

The remaining configuration options are not specific to the plugin use case and merely mirror the features of the library API, apart from differences in the syntax/mechanics which with they are applied, which are reflected in the corresponding code examples. For this reason this section only gives a very brief overview while linking to the relevant sections in the other chapters.

laikaTheme Setting

This setting allows to configure the built-in Helium theme or a 3rd-party theme you might use. You can also use it to specify an empty theme if you want to put all templates and styles right into your input directory.

Example for applying a few Helium settings:

```
laikaTheme := Helium.defaults
  .all.metadata(
    title = Some("Project Name"),
    language = Some("de"),
  )
  .epub.navigationDepth(4)
  .pdf.navigationDepth(4)
  .build
```

Setting an empty theme:

```
laikaTheme := Theme.empty
```

If you do not use this property Laika will run with the default settings of the Helium theme, meaning your site and e-books will look exactly like this documentation, including all color and font choices.

Since the theme offers a lot of configuration options it has its own dedicated chapter **Theme Settings**.

To just give a brief overview, those settings allow you to configure:

- The **Fonts** to embed in EPUB and PDF files or to link in HTML files.

- The **Colors** for the main theme and for syntax highlighting.

- Several aspects of the theme's **Layout**, like column widths, block spacing or PDF page sizes.

- **Metadata** (title, authors, description, language, etc.) to include in the generated site or e-books.

- Adding **Favicons** or custom links to the **Top Navigation Bar**

- Adding an optional **Download Page** for the site's content as EPUB or PDF

- Providing logos, text and links for an optional **Website Landing Page**.

- Configure **Cover Images for E-books** or **Auto-Linking CSS & JS Files**.

laikaConfig Setting

A builder-style API for basic configuration options grouped in a single sbt setting. These are configuration options not tied to the default theme and are available even when using a 3rd party theme or no theme at all.

Example:

```
laikaConfig := LaikaConfig.defaults.strict.withRawContent
```

- **Strict mode**: Disables all non-standard extensions Laika adds to the supported markup formats, like directives.

- **Raw Content**: Enables the inclusion of raw sections of the target format in markup files, e.g. snippets of verbatim HTML embedded in Markdown files. By default this is disabled.

- **Character Encoding**: Sets the character encoding of input and output files, the default is UTF-8.

- **Error Handling**: Specify log levels or switch to "visual debugging", where recoverable errors are rendered in the page context where they occur instead of causing the transformation to fail.

- **User-Defined Variables**: Define variables globally that you can refer to in templates and markup files with substitution references.

laikaExtensions Setting

Register implementations of the `ExtensionBundle` API, either provided by the library or extensions you created yourself.

Example:

```
laikaExtensions := Seq(GitHubFlavor, SyntaxHighlighting)
```

- **Overriding Renderers**: adjust the rendered output for specific AST node types.

- **AST Rewriting**: transform the document AST between parsing and rendering.

- **Implementing Directives**: install custom directives.

- Or use any other hook in **The ExtensionBundle API**).

Configuring Input and Output

Most of these setting are introduced in the sections above.

- `Laika / sourceDirectories` default `Seq(sourceDirectory.value / "docs")`, usually `Seq("src/docs")`. Specifies one or more source directories to process to be merged into a tree with a single root. See **Preparing Content** for details.

- `Laika / target` - default `target.value / "docs"`, usually `Seq("target/docs")`. Specifies the directory where the plugin should generate the site. See **Generating a Site** for details.

- `Laika / excludeFilter` - default `HiddenFileFilter`
  Specifies files to exclude from processing. Note that Laika ignores any `includeFilter` you set, as the tool needs more than a simple yes/no decision for deciding how to process files.

- `laikaDocTypeMatcher` extension: Where a simple `excludeFilter` is not sufficient you can customize the way Laika determines the document type.

```
laikaExtensions += laikaDocTypeMatcher {
  case Root / "templates" / _ => DocumentType.Template
  case Root / "images" / _ => DocumentType.Static
}
```

The list of pattern matches can be incomplete, they will be added to the built-in default matches.

Freely Composing Inputs

If you need additional flexibility instead of just configuring one or more input directories, e.g. when there is a need to generate content on-the-fly before starting the transformation, you can use the `laikaInputs` setting. This setting completely overrides any value set with `Laika / sourceDirectories`.

```
val inputs = InputTree[F]
  .addDirectory("/path-to-my/markup-files")
  .addDirectory("/path-to-my/images", Root / "images")
  .addClasspathResource("my-templates/
default.template.html", DefaultTemplatePath.forHTML)
  .addString(generateStyles(), Root / "css" / "site.css")
```

In the example above we specify two directories, a classpath resource and a string containing CSS generated on the fly. By default directories get merged into a single virtual root, but in the example we declare a mount point for the second directory, which causes the content of that directory to be assigned the corresponding logical path.

> ℹ️
>
> Always keep in mind that declaring inputs and outputs are the only places in the Laika universe where you'd ever use concrete file system paths. Beyond this configuration step you are entirely within Laika's virtual path abstraction and refer to other resources by their virtual path. This is true for linking, using image resources, specifying templates to use in configuration headers, and so on. It means that everything you can refer to in your markup files needs to be included in the input composition step.

The `InputTreeBuilder` API gives you the following options:

- Add entire directories and optionally specify a "mount point" at which they should be linked in the virtual tree.

- Specify individual files or classpath resources.

- Add in-memory string which will enter the parsing pipelines like file resources.

- Add in-memory AST nodes which will by-pass the parsing step and added directly to the other document AST obtained through parsing.

When generating input on the fly it is usually a question of convenience or reducing boilerplate whether you choose to generate a string for parsing or the AST directly.

For the complete API see **InputTreeBuilder**.

Settings for the laikaSite task

Finally there are three boolean flags that only affect the laikaSite task.

- `laikaIncludeAPI` - default `false` - see **Including Scaladoc**.

- `laikaIncludeEPUB` - default `false` - see **Including EPUB and PDF**.

- `laikaIncludePDF` - default `false` - see **Including EPUB and PDF**.

Inspecting Laika's Configuration

Run `show laikaDescribe` to get a formatted summary of the active configuration, installed extension bundles and lists of input and output files.

Example:

```
Parser(s):
  Markdown
  reStructuredText
Renderer:
  AST
Extension Bundles:
  Laika's Default Extensions (supplied by library)
  Laika's directive support (supplied by library)
  Laika's built-in directives (supplied by library)
  Default Syntax Highlighters for Code (supplied by library)
  Document Type Matcher for Markdown (supplied by parser)
  Default extensions for reStructuredText (supplied by parser)
  Support for user-defined reStructuredText directives (supplied by parser)
  Standard directives for reStructuredText (supplied by parser)
  Document Type Matcher for reStructuredText (supplied by parser)
Settings:
  Strict Mode: false
  Accept Raw Content: false
  Render Formatted: true
Sources:
  Markup File(s)
    File '/dev/project/src/intro.md'
    File '/dev/project/src/details.md'
  Template(s)
    In-memory string or stream
  Configuration Files(s)
    -
  CSS for PDF
    -
  Copied File(s)
```

```
    File '/dev/project/src/main.css'
  Root Directories
    -
Target:
  Directory '/dev/project/target/docs'""""
```

Library API

Laika's API is fully referentially transparent, does not throw exceptions and avoids any kind of runtime reflection. Its core module is also available for Scala.js. The library only has a very limited number of dependencies and does not require the installation of any external tools.

This chapter only covers the topics which are specific to using Laika without its sbt plugin, while most other parts of the manual apply to both use cases, library and plugin.

Dependencies

Which of the modules you need to add to the build depends on the functionality you intend to use.

If you want to stick to pure transformations from string to string and don't need file or stream IO or any of the binary output formats like EPUB or PDF, you are fine with just using the `laika-core` module:

```
libraryDependencies += "org.planet42" %% "laika-core" % "0.16.1"
```

This module is also 100% supported for Scala.js, so you can alternatively use the triple `%%%` syntax if you want to cross-build for Scala.js and the JVM:

```
libraryDependencies += "org.planet42" %%% "laika-core" % "0.16.1"
```

If you want to add support for file and stream IO and/or output in the EPUB format, you need to depend on the `laika-io` module instead:

```
libraryDependencies += "org.planet42" %% "laika-io" % "0.16.1"
```

This depends on `laika-core` in turn, so you always only need to add one module as a dependency and will get the rest via transitive dependencies. No module apart from `laika-core` is available for Scala.js, so you are in JVM land here.

Finally PDF support comes with its own module as it adds a whole range of additional dependencies:

```
libraryDependencies += "org.planet42" %% "laika-pdf" % "0.16.1"
```

Again, this builds on top of the other modules, so adding just this one dependency is sufficient.

Anatomy of the API

The library API for creating parsers, renderers or transformers, while aiming to be intuitive and simple, also exposes one of the main design goals of Laika: that everything is meant to be pluggable.

The following example for creating a pure transformer shows the main building blocks:
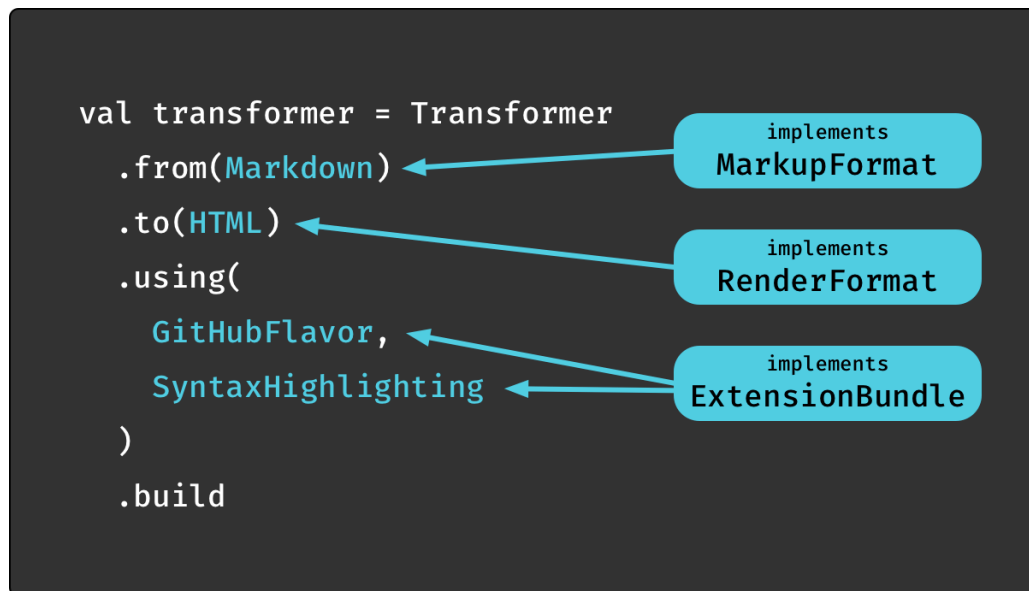
- All inputs need to support the `MarkupFormat` contract. Supported by the core library are `Markdown` and `ReStructuredText`.

- Output formats implement the `RenderFormat[FMT]` trait. The type parameter represents the formatter API that renderer overrides for specific AST nodes can talk to, which is different depending on the format. The library supports the formats `HTML`, `EPUB`, `PDF`, `XSLFO` and `AST` out of the box. The latter is a debug output that renders the document AST in a formatted structure.

- Finally, the optional `using` hook in the API accepts anything that implements `ExtensionBundle`. This API has hooks for all phases of the transformation process, parsing, AST transformation and rendering and is Laika's main extension point. Shown above are `GitHubFlavor` and `SyntaxHighlighting`, two bundles provided by the library that can be enabled on demand.

If you require support for file/stream IO, templating or binary output formats, the `laika-io` module expands on the core API to add this functionality:

```
val transformer = Transformer          implements
  .from(Markdown)                      MarkupFormat
  .to(HTML)                            implements
  .using(                              RenderFormat
    GitHubFlavor,                      implements
    SyntaxHighlighting                 ExtensionBundle
  )                                    instance of
  .io(blocker)                         cats.effect.Blocker
  .parallel[IO]                        F[_]: Sync: ContextShift
  .withTheme(Helium.defaults.build)
  .build                               ThemeProvider
```
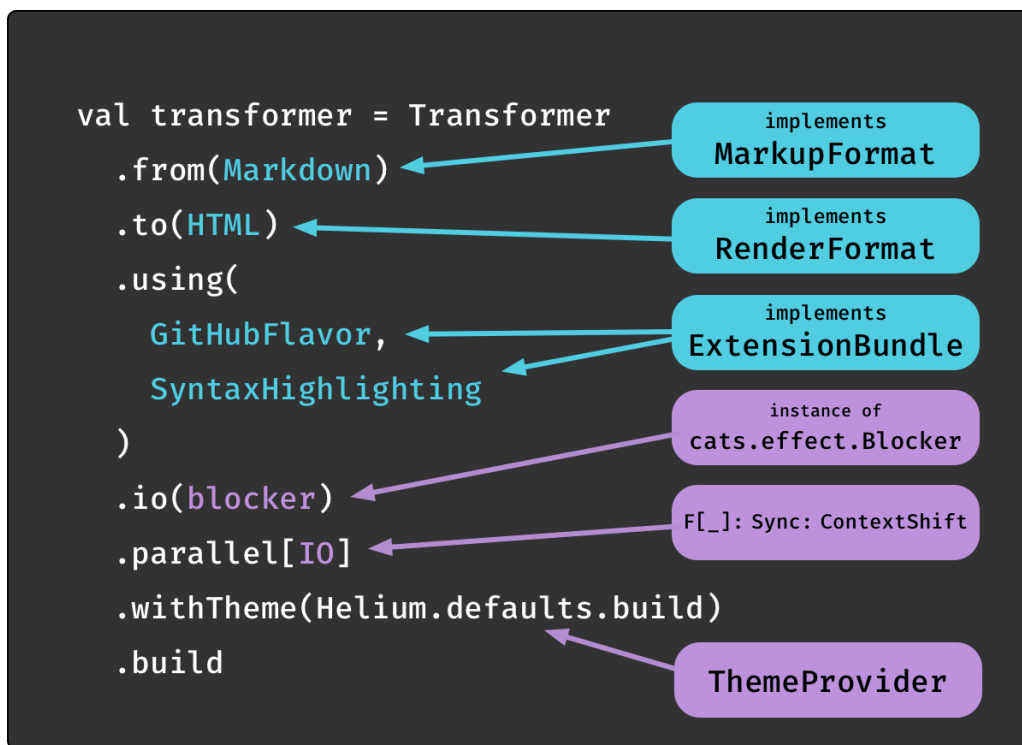
The blue elements of the API are identical to the pure transformer. The new `io` method used above becomes available with `import laika.io.implicits._`.

This API introduces a dependency on `cats-effect` which is used to model the effectful computations. You can use it with any effect that supports the `cats-effect` type classes, like cats-IO, Monix or Zio.

- The blocker passed in the example gives you control over which `ExecutionContext` blocking IO is performed in.

- The call to `parallel[IO]` builds a transformer that lets you specify entire directories as input and allows to instruct the library which effect type to use (cats-IO in this example). There are two more variants, `sequential`, that processes inputs sequentially as well as an overload for `parallel` that allows to specify the level of parallelism.

Using a `Transformer` is the most convenient option when you want to go directly from the raw input format (text markup) to the final output format. There are also `Parser` and `Renderer` instances that only do one half of the transformation.

All types, `Transformer`, `Parser` and `Renderer` come in two flavors each. A simple, pure API for dealing with strings as input and output and a parallel, effectful API for processing entire directories with templates and configuration files in addition to the usual text markup input.

Transforming Strings

Transforming in-memory input is the simplest form of transformation, and works for Markdown and reStructuredText as input, and for HTML as output. EPUB and PDF both require additional modules and are described later in this chapter.

For most cases where you don't use any of the customization hooks, you should be fine with just these imports:

```
import laika.api._
import laika.format._
```

As the next step you can then create transformer instances, which in most cases you'll want to reuse with different inputs to reduce the memory footprint and initialization overhead. The instances are immutable and thread-safe.

Example for creating a transformer for Markdown, including the GitHub-Flavor extensions:

```
val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)
  .build
```

Example for creating a transformer for reStructuredText:

```
val transformer = Transformer
  .from(ReStructuredText)
  .to(HTML)
  .build
```

In addition to specifying markup and output formats and extensions, there are many other customization options in the builder step immediately before calling `build`. They are identical for the pure and IO parser and are summarized in the **Configuration** section.

Finally you can use the transformer with text markup as input:

```
val result = transformer.transform("hello *there*")
```

The result is of type `Either[ParserError, String]`.

Setup for Effectful Transformations

In case you want to go beyond pure string transformations, you need to switch your dependency to the `laika-io` module as shown in the **Dependencies** section.

This dependency will add file/stream IO, theme support, templating and the ability to process entire directories.

This module depends on cats-effect, and models all side effects in an abstract effect, so that you can use it with cats IO, Monix or Zio.

With the dependency in place you also need to add a third import to those you used for pure transformations:

```
import laika.api._
import laika.format._
import laika.io.implicits._
```

The remainder of the setup depends on whether you are **Using cats.IO, Monix or Zio** or whether you build **Applications without Effect Library**.

Using cats.IO, Monix or Zio

When your application is already using one of the effect libraries compatible with the cats-effect type class hierarchy, integrating Laika will be seamless.

The exact mechanics of your Laika transformer setup will depend on which of the aforementioned libraries you are using and whether you keep all of you code in an abstract `F[_]` instead of coding against a concrete effect type.

The following example assumes the use case of an application written around abstract effect types and using `IOApp` from cats.IO for initialization:

```
def createTransformer[F[_]: Async: ContextShift]
    (blocker: Blocker): Resource[F, TreeTransformer[F]] =
  Transformer
    .from(Markdown)
    .to(HTML)
    .using(GitHubFlavor)
    .io(blocker)
    .parallel[F]
    .build
```

The setup method above can then be used inside `IOApp` initialization logic:

```scala
object MyApp extends IOApp {

  def run(args: List[String]) = {
    Blocker[IO].use { blocker =>
      val transformer = createTransformer[IO](blocker)
      // other setup code
    }.as(ExitCode.Success)
  }
}
```

This way Laika gives full control over the `ExecutionContext` in which the blocking IO and CPU-bound operations are performed.

Setup for other libraries would be similar, Monix for example comes with a `TaskApp` which is similar to `IOApp`.

Applications without Effect Library

When using other stacks like Akka HTTP or the Play framework, you need to bridge the gap between the worlds of cats-effect and the surrounding toolkit, often centering around the Future API (which, in contrast to cats-effect, is not referentially transparent).

First, to create a transformer with the setup method created in the previous example, you need to create instances of `ContextShift` and `Blocker` yourself:

```scala
implicit val cs: ContextShift[IO] =
  IO.contextShift(ExecutionContext.global)

val blocker = Blocker.liftExecutionContext(
  ExecutionContext.fromExecutor(Executors.newCachedThreadPool())
)

val transformer = createTransformer(blocker)
```

The resulting instance can then be used to describe a transformation:

```scala
val result: IO[String] = transformer.use {
  _.fromDirectory("docs")
   .toDirectory("target")
   .transform
}
```

The resulting `IO` instance is only a description of a program, as it starts evaluation lazily, in contrast to the Scala SDK's `Future` which starts processing eagerly.

For actually running the effect and producing a result, you have several options depending on the stack you are using. One common scenario is a toolkit like Akka HTTP or Play where you execute a route that is expected to return a `Future`. This can achieved by a simple translation:

```
val futureResult: Future[String] = result.unsafeToFuture()
```

Other options not involving `Future` are either blocking, synchronous execution:

```
val syncResult: String = result.unsafeRunSync()
```

Or asynchronous execution based on a callback:

```
result.unsafeRunAsync {
  case Left(throwable) => handleError(throwable)
  case Right(result)   => handleResult(result)
}
```

Do not get too hung up on the scary sound of all these `unsync...` methods. The one kind of safety you are loosing when using them is referential transparency. But if you are using a `Future`-based API for example, your program is not referentially transparent anyway.

Entire Directories as Input

The effectful transformer is the most powerful variant and also the one that is the basis for the sbt plugin. It expands the functionality beyond just processing markup files to also parsing templates and configuration files as well as copying static files over to the target directory.

```
val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)
  .io(blocker)
  .parallel[IO]
  .build
```

The above transformer can then be used to process a directory of markup, template and configuration files:

```
val res: IO[RenderedTreeRoot[IO]] = transformer.use {
  _.fromDirectory("src")
   .toDirectory("target")
   .transform
}
```

The `target` directory is expected to exist, while any required subdirectories will be automatically created during rendering. There will be one HTML file for each input file in the same directory layout.

If you replace the `HTML` renderer in the example above with one of the binary formats (`EPUB` or `PDF`), the API will be slightly different, as the `toDirectory` option is replaced by `toFile`. This is because these formats always produce a single binary result, merging all content from the input directories into a single, linearized e-book:

```
val res: IO[Unit] = transformer.use {
  _.fromDirectory("src")
   .toFile("output.epub")
   .transform
}
```

Merging Multiple Directories

All previous examples read content from the single input directory. But you can also merge the contents of multiple directories:

```
val res: IO[RenderedTreeRoot[IO]] = transformer.use {
  _.fromDirectories("markup", "theme")
   .toDirectory("target")
   .transform
}
```

This adds some additional flexibility, as it allows, for example, to keep reusable styles and templates separately. This flexibility is possible as Laika is not tied to the file system, but instead provides a **Virtual Tree Abstraction**.

Freely Composing Inputs

If you need even more flexibility instead of just configuring one or more input directories, e.g. when there is a need to generate content on-the-fly before starting the transformation, you can use the `InputTree` builder.

```
val inputs = InputTree[F]
  .addDirectory("/path-to-my/markup-files")
  .addDirectory("/path-to-my/images", Root / "images")
  .addClasspathResource("templates/default.html", DefaultTemplatePath.forHTML)
  .addString(generateStyles(), Root / "css" / "site.css")
```

In the example above we specify two directories, a classpath resource and a string containing CSS generated on the fly. By default directories get merged into a single virtual root, but in the example we declare a mount point for the second directory, which causes the content of that directory to be assigned the corresponding logical path.

> ℹ️
>
> Always keep in mind that declaring inputs and outputs are the only places in the Laika universe where you'd ever use concrete file system paths. Beyond this configuration step you are entirely within Laika's virtual path abstraction and refer to other resources by their virtual path. This is true for linking, using image resources, specifying templates to use in configuration headers, and so on. It means that everything you can refer to in your markup files needs to be included in the input composition step.

The `InputTreeBuilder` API gives you the following options:

- Add entire directories and optionally specify a "mount point" at which they should be linked in the virtual tree.

- Specify individual files or classpath resources.

- Add in-memory string which will enter the parsing pipelines like file resources.

- Add in-memory AST nodes which will by-pass the parsing step and added directly to the other document AST obtained through parsing.

When generating input on the fly it is usually a question of convenience or reducing boilerplate whether you choose to generate a string for parsing or the AST directly.

For the complete API see **InputTreeBuilder**.

The customized input tree can then be passed to the transformer:

```
val res: IO[RenderedTreeRoot[IO]] = transformer.use {
  _.fromInputTree(inputs)
   .toDirectory("target")
   .transform
}
```

Preparing Content

Laika does not have any special directories and content can be nested in sub-directories down to arbitrary levels. For more details on how to organize content, see **Directory Structure**.

Transformers and Parsers distinguish between the following file types:

- **Markup Files**: Files with the extensions `.md`, `.markdown` or `.rst` (depending on the specified input format) will be parsed and rendered to the target in the same directory structure and with the same file names apart from the suffix, which will be replaced depending on the output format (e.g. `.html`).

- **Configuration Files**: Each directory can contain an optional `directory.conf` file for specifying things like navigation order or chapter title. See **Configuration Files** for details.

- **Template Files**: You can provide a default template per directory with the name `default.template.<suffix>` with the suffix matching the output format (e.g. `.html`). They will also be applied to sub-directories, unless overridden. Default templates in one of your input directories always override default templates provided by themes. You can also add additional templates with the name pattern `*.template.<suffix>`, which will only be applied when a markup document explicitly refers to them in its configuration header.

- **Static Files**: All other files, like CSS, JavaScript, images, etc., will be copied over to the target in the same directory structure and with identical file names.

Separate Parsing and Rendering

So far all examples in this chapter showed the use of a transformer instance, which takes you all the way from a raw input file to the target format, e.g. from Markdown to HTML.

But in some cases you may just want to execute one half of the process:

- If the input is generated in-memory. In this case there is no need to generate text markup, you can directly produce the document AST and pass it to a `Renderer`.

- If you want to process the document AST produced by a `Parser` by other means than rendering one of the supported output formats.

- If you want to render the same input to multiple output formats. In this case building a `Transformer` for all output formats is inefficient as each of them would parse the same input again. Instead you can create a single `Parser` for the input and in addition one `Renderer` for each output format you want to generate.

In fact if you examine the implementations of the various pure and effectful transformer types, you'll notice that it is mostly trivial: delegating to the underlying `Parser` and `Renderer` and just piping the result from one to the other.

The following code example demonstrates the third scenario listed above: Rendering the same input to multiple output formats.

First we create a parser that reads from a directory:

```
val parserRes = MarkupParser
  .of(Markdown)
  .using(GitHubFlavor)
  .io(blocker)
  .parallel[IO]
  .build
```

Next we create the renderers for the three output formats:

```
val htmlRendererRes = Renderer.of(HTML).io(blocker).parallel[IO].build
val epubRendererRes = Renderer.of(EPUB).io(blocker).parallel[IO].build
val pdfRendererRes  = Renderer.of(PDF).io(blocker).parallel[IO].build
```

Since all four processors are a cats-effect `Resource`, we combine them into one:

```
val allResources = for {
  parser <- parserRes
  html <- htmlRendererRes
  epub <- epubRendererRes
  pdf <- pdfRendererRes
} yield (parser, html, epub, pdf)
```

Finally we define the actual transformation by wiring the parser and the three renderers:

```
val transformOp: IO[Unit] = allResources.use {
  case (parser, htmlRenderer, epubRenderer, pdfRenderer) =>
    parser.fromDirectory("source").parse.flatMap { tree =>
      val htmlOp = htmlRenderer.from(tree.root).toDirectory("target").render
      val epubOp = epubRenderer.from(tree.root).toFile("out.epub").render
      val pdfOp = pdfRenderer.from(tree.root).toFile("out.pdf").render
      (htmlOp, epubOp, pdfOp).parMapN { (_, _, _) => () }
    }
}
```

We are using cats' `parMapN` here to run the three renderers in parallel.

The `tree` instance passed to all renderers is of type `DocumentTreeRoot`. If necessary you can use its API to inspect or transform the tree before rendering. See **The Document AST** for details.

The sample code in this scenario showed the effectful variant of the `Parser` and `Renderer` types, but just as the `Transformer` they exist in the other flavor as well: a pure variant as part of the `laika-core` module.

Configuration

All the examples in this chapter only scratched the surface of Laika's API, focusing on the basics like specifying input and output. But since Laika is designed to be fully customizable and serve as a toolkit for creating toolkits, there is whole range of options available for customizing and extending its built-in functionality.

Most of these configuration options are not specific to the library API use case and apply to the sbt plugin as well, apart from differences in the syntax/mechanics which with they are applied, which are reflected in the corresponding code examples. For this reason this section only gives a very brief overview while linking to the relevant sections in the other chapters.

Theme Configuration

Configuration for the built-in Helium theme or a 3rd-party theme can be passed to the transformer builders. You can also specify an empty theme if you want to put all templates and styles right into your input directory.

Example for applying a few Helium settings:

```
val theme = Helium.defaults
  .all.metadata(
    title = Some("Project Name"),
    language = Some("de"),
  )
  .epub.navigationDepth(4)
  .pdf.navigationDepth(4)
  .build

val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)
  .io(blocker)
  .parallel[IO]
  .withTheme(theme)
  .build
```

Setting an empty theme:

```
laikaTheme := Theme.empty
```

If you do not explicitly pass a theme configuration Laika will run with the default settings of the Helium theme, meaning your site and e-books will look exactly like this documentation, including all color and font choices.

Since the default Helium theme offers a lot of configuration options it has its own dedicated chapter **Theme Settings**. They are specific to the Helium theme, when using a 3rd-party theme you need to consult their documentation for setup options.

To just give a brief overview, those settings allow you to configure:

- The **Fonts** to embed in EPUB and PDF files or to link in HTML files.

- The **Colors** for the main theme and for syntax highlighting.

- Several aspects of the theme's **Layout**, like column widths, block spacing or PDF page sizes.

- **Metadata** (title, authors, description, language, etc.) to include in the generated site or e-books.

- Adding **Favicons** or custom links to the **Top Navigation Bar**

- Adding an optional **Download Page** for the site's content as EPUB or PDF

- Providing logos, text and links for an optional **Website Landing Page**.

- Configure **Cover Images for E-books** or **Auto-Linking CSS & JS Files**.

Other Settings

These other settings are available independent from the theme in use:

- **Strict mode**: Disables all non-standard extensions Laika adds to the supported markup formats, like directives.

- **Raw Content**: Enables the inclusion of raw sections of the target format in markup files, e.g. snippets of verbatim HTML embedded in Markdown files. By default this is disabled.

- **Character Encoding**: Sets the character encoding of input and output files, the default is UTF-8.

- **Error Handling**: Specify log levels or switch to "visual debugging", where recoverable errors are rendered in the page context where they occur instead of causing the transformation to fail.

- **Laika Extensions**: Use the library's customization options to override renderers for specific AST nodes (**Overriding Renderers**), transform the document AST before rendering (**AST Rewriting**), install custom directives (**Implementing Directives**) or use some of the lower level hooks in (**The ExtensionBundle API**).

- **Basic Configuration Options**: Setting **Strict Mode**, including **Raw Content**, setting the **Character Encoding**, controlling **Error Handling** and specifying **User-Defined Variables**.

- **Inspecting Laika's Configuration**: Run the `describe` method on the IO-based transformers, parsers and renderers to get a formatted summary of the active configuration, installed extension bundles and lists of input and output files.

Configuration

Laika has a rich set of features and keeps almost every module customizable or even replaceable, and as a consequence there is a lot to configure.

This chapter puts the focus on the basic configuration options that you are most likely to use on a day-to-day basis.

If you want to dig deeper and tweak or extend the way Laika parses or renders, see the intro sections to **Customizing Laika** and **Extending Laika** for an overview.

Theme Settings

Laika comes with a lightweight default theme called Helium. Since the theme offers a lot of configuration options it has its own dedicated chapter **Theme Settings**.

To just give a brief overview, those settings allow you to configure:

- The **Fonts** to embed in EPUB and PDF files or to link in HTML files.

- The **Colors** for the main theme and for syntax highlighting.

- Several aspects of the theme's **Layout**, like column widths, block spacing or PDF page sizes.

- **Metadata** (title, authors, description, language, etc.) to include in the generated site or e-books.

- Adding **Favicons** or custom links to the **Top Navigation Bar**

- Adding an optional **Download Page** for the site's content as EPUB or PDF

- Providing logos, text and links for an optional **Website Landing Page**.

- Configure **Cover Images for E-books** or **Auto-Linking CSS & JS Files**.

Navigation

These settings also have their dedicated **Navigation** chapter, including configuration examples, therefore we'll just give a brief overview of available configuration options here and link to the relevant sections.

- **Global Link Definitions** help to avoid repetition by mapping URLs to an id in configuration and making them available for use with "native" markup link syntax (e.g. `link to [my-globally-defined-id]` in Markdown).

- **Linking to API Documentation** describes the `@:api` shortcut for defining links to API documentation, which requires setting the relevant base URLs in the configuration.

- **Disabling Validation** is a configuration option for excluding some directories/paths from validation, e.g. in cases where you know that a directory will be populated by some external tool. In its default setup,

a Laika transformation fails when an internal link points to a document that does not exist in the input directory.

- E-books (EPUB or PDF) have additional configuration options for navigation:

  - **Book Navigation** allows to specify the depth of the auto-generated navigation.

  - The location of **Cover Images for E-books** can be configured, too.

  - And **Metadata** can be specified so that it will be converted to a format compatible with the corresponding readers.

Run Modes

These settings are available in the library API and the sbt plugin.

Strict Mode

Strict mode switches off all extensions and only uses features defined in the spec of the markup languages.

This might be useful if you develop applications with user input for example, and want to ensure that only standard markup features are exposed to users.

Markup extensions enabled by default are:

- Custom Directives - either the built-in **Standard Directives** or user-defined as shown in **Implementing Directives**.

- Support for HOCON configuration headers in markup files.

- Support for **Substitution Variables** in markup in the form of `${some.key}`.

To disable all these extensions you can use the `strict` flag:

```
laikaConfig := LaikaConfig.defaults.strict
```

Raw Content

Raw content is the term Laika uses for any markup syntax that allows the inclusion of content in the output format, e.g. HTML.

This is disabled by default, even if it is part of the original text markup specification as in the case of Markdown.

This is to generally discourage to tie markup input to a concrete output format, as Laika supports multiple formats like HTML, EPUB and PDF and an API where users can easily add their own format. Markup files containing raw HTML could not be used for those.

Secondly, when the markup originates from user input in a web application, it would not be safe to use without additional filtering. By default Laika does not filter any tags, not even `<script>` tags, but whitelist filtering can be added fairly easily through Laika's customization hooks like **AST Rewriting** or **Overriding Renderers**.

You can enable verbatim HTML and other raw formats explicitly in the configuration:

```
laikaConfig := LaikaConfig.defaults.withRawContent
```

Character Encoding

The default encoding in Laika is UTF-8.

When you need to work with different encodings you can override the default:

```
laikaConfig := LaikaConfig.defaults.encoding(Codec.ISO8859)
```

Error Handling

Text markup parsers are usually very resilient. For any input they cannot make sense of they fall back to rendering it back as raw text. If, for example you start an emphasized span with an `'*'`, but then never close the span, it will simply be treated as a literal and will not trigger an error.

But there are two groups of errors that do cause a transformation to fail.

Error Types

- **Unrecoverable errors**: for example I/O errors where files cannot be read or directories cannot be written to, or invalid configuration, e.g. in HOCON configuration headers, which is also treated as fatal, since such a configuration section usually drives the behaviour or rendering of other nodes.

- **Invalid nodes**: these are "local" errors, which are confined to a single AST node. This may be a reference to a section, document or footnote that does not exist or a directive with invalid or missing attributes.

The handling of the second kind of error can be controlled via configuration.

Default Behaviour

AST nodes that capture errors carry an error message, a severity and a fallback node to use if configuration would allow to proceed with rendering.

Laika comes with two configuration options that control how these nodes are dealt with:

- `failOnMessages` controls the minimum severity that causes a transformation to fail, the default is `Error`.

- `renderMessages` controls the minimum severity that causes a message to be included in the rendered output, the default is `None`.

This means, by default the presence of one or more invalid nodes in the AST of a document causes the transformation to fail, with all their message collected in the returned error type.

Visual Debugging

In some cases you may prefer to examine the errors in the rendered content, or you are processing user input where it is more helpful to display them in context than just as a list of error messages.

You can achieve this by basically flipping the two default values in the configuration:

```
laikaConfig := LaikaConfig.defaults
  .failOnMessages(MessageFilter.None)
  .renderMessages(MessageFilter.Error)
```

Now rendering proceeds even with invalid nodes and they will be rendered in the location of the document they occurred in. In HTML output these node are rendered as a span with the class `runtime-message` and a second class for the level (`warning`, `error`, etc.), so that their display can be controlled via CSS.

The AST Renderer

If you need even more low-level insight, you can use the included AST renderer to get a formatted output of the entire AST structure.

Longer plain text spans are truncated in this output format to put the emphasis on the structure of the document.

Create a file with the content `some *text* example` in the input directory and run:

```
laikaAST
```

This task is a shortcut for `laikaGenerate ast`

The output for the small example above will have the following format:

```
RootElement - Blocks: 1
. Paragraph - Spans: 3
. . Text - 'some '
. . Emphasized - Spans: 1
. . . Text - 'text'
. . Text - ' example'
```

User-Defined Variables

Finally, the entire machinery for passing configuration around can be used by users, too.

You can define variables in any of the following scopes:

- Programmatically in global configuration

- Per directory with HOCON in a file named `directory.conf`

- Per markup document in a HOCON configuration header

This is an example for defining two variables globally:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue("version.latest", "2.4.6")
  .withConfigValue("license", "Apache 2.0")
```

These values can then be accessed via **Substitution Variables** in templates or in markup files:

```
The latest release is ${version.latest}.
It's released under the ${license} license.
```

If you define them in a narrower scope and not globally, they won't be available outside of that scope.

There are three namespaces for variable keys which are used by the library, `laika.*`, `cursor.*` and `helium.*`. As long as you avoid these three namespaces, you can freely use any configuration keys.

Directory Structure

Laika does not have any special directories and content can be nested in sub-directories down to arbitrary levels.

When you are using the sbt plugin or the tree transformer from the library API that accepts directories as input, Laika supports additional file types and navigation features on top of just parsing text markup. This section describes the supported **Document Types**, how **Auto-Generated Navigation** works with directory input and how the **Virtual Tree Abstraction** decouples the logic from the file system.

This set of functionality is not available when you are using a transformer or parser from the library API that expects a single input (file, stream or string) for processing.

Document Types

The library distinguishes between the following file types:

Markup Files

Files with the extensions `.md`, `.markdown` or `.rst` will be parsed and rendered to the target in the same directory structure and with the same file names apart from the suffix, which will be replaced depending on the output format (e.g. `.html`).

Apart from standard markup syntax, markup files in Laika can also contain the following non-standard elements (unless running in strict mode which disables all extensions):

- A configuration header enclosed between `{%` and `%}` in HOCON format

- A document title for representing the page in navigation structures. The title is obtained from the first header element in the page unless overridden with the `laika.title` key in the configuration header.

- Sections, determined by the hierarchy of headers coming after the title, with auto-assigned ids based on a pluggable slug builder. See **Customization Hooks** for details. Sections can be part of auto-generated navigation structures and can conveniently be linked to by header text with "native" markup link syntax **Linking by Section Headline**.

- Directives, which extend the text markup's functionality, either one of the built-in **Standard Directives** or your own, custom implementations.

Title Documents

Each directory can contain an optional title document. It is recognized by the file name pattern `README.<suffix>`, e.g. `README.md`. When rendering the name will change to `index.<suffix>`, e.g. `index.html`.

These defaults have been chosen so that markup files appear below directory navigation on GitHub and rendered HTML files can serve as index pages. The names can be overridden in Laika's global configuration (you need to omit the suffix, as it'll work with all formats):

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LaikaKeys.titleDocuments.inputName, "title")
  .withConfigValue(LaikaKeys.titleDocuments.outputName, "title")
```

Title Documents will also render on a level above the other chapter documents in the navigation, (see **Auto-Generated Navigation** for details) and will be linked in **Breadcrumbs** components.

Configuration Files

Each directory can contain an optional `directory.conf` file for configuring the directory (and its sub-directories). The expected format of the document is HOCON.

Amongst the most common things you would configure in this kind of file are titles and navigation order:

```
laika {

  title = Fruits

  navigationOrder = [
    apples.md
    oranges.md
    strawberries.md
    some-subdirectory
    other-subdirectory
  ]
}
```

The `navigationOrder` node in the example above contains both markup files and sub-directories. They would appear in this order in tables of contents and the same order would be applied when autonumbering.

The default ordering, when not provided explicitly, is alphabetical.

Template Files

You can provide a default template per directory with the name `default.template.<suffix>`, where the suffix matches the output format (e.g. `.html`). They will also be applied to sub-directories, unless overridden. Default templates in one of your input directories always override default template provided by themes.

Additionally you can add templates with the name pattern `*.template.<suffix>`, which will only be applied when a markup document explicitly refers to them in its configuration header.

For more details on the template engine, see the chapter **Creating Templates**.

Static Files

All other files, like CSS, JavaScript, images, etc., will be handled depending on the output format:

- **HTML**: When generating sites, static files will be copied over to the target in the same directory structure and with identical file names.

- **EPUB**: Static files will be integrated into the EPUB container, in the same directory structure and with identical file names. See **E-Books (EPUB & PDF)** for supported file types and other options.

- **PDF**: When generating PDF only assets actually used and referred to by markup pages or templates will be embedded in the final binary file. See **E-Books (EPUB & PDF)** for supported file types and other options.

Auto-Generated Navigation

The directory structure in a Laika project is not only a way to organize files in a workspace, the structure will also be used by various features of the library that auto-generate navigation structures.

- **EPUB & PDF**: The supported E-book formats will include auto-generated navigation. The hierarchy of the navigation will match the directory structure of your inputs and the configured navigation order. The actual content will get flattened into a linear book flow, with title documents becoming the first section in each chapter. See **E-Books (EPUB & PDF)** on how to configure navigation depth and other options.

- **Web Sites**: For HTML output you can use directives like **The navigationTree Directive** in templates to auto-generate a navigation structure from the root of your input tree or any other specific sub-node. The default Helium theme also provides a main navigation tree in the left sidebar and page navigation on the right out of the box.

The presence of title documents would determine how exactly chapter title are rendered in the navigation structure.

Example for a structure with title documents

```
                                    About Spices              Generated
                                    ===========              Navigation:
    📁 spices
                                    Text...              About Spices
        📄 README.md
        📄 saffron.md                                       Saffron
        📄 cinnamon.md                                      Cinnamon
        📄 nutmeg.md                                        Nutmeg
    📁 herbs                                             About Herbs
        📄 README.md                                        Parsley
        📄 parsley.md                                       Coriander
        📄 coriander.md              About Herbs            Basil
                                    ===========
        📄 basil.md
                                    Text...
```
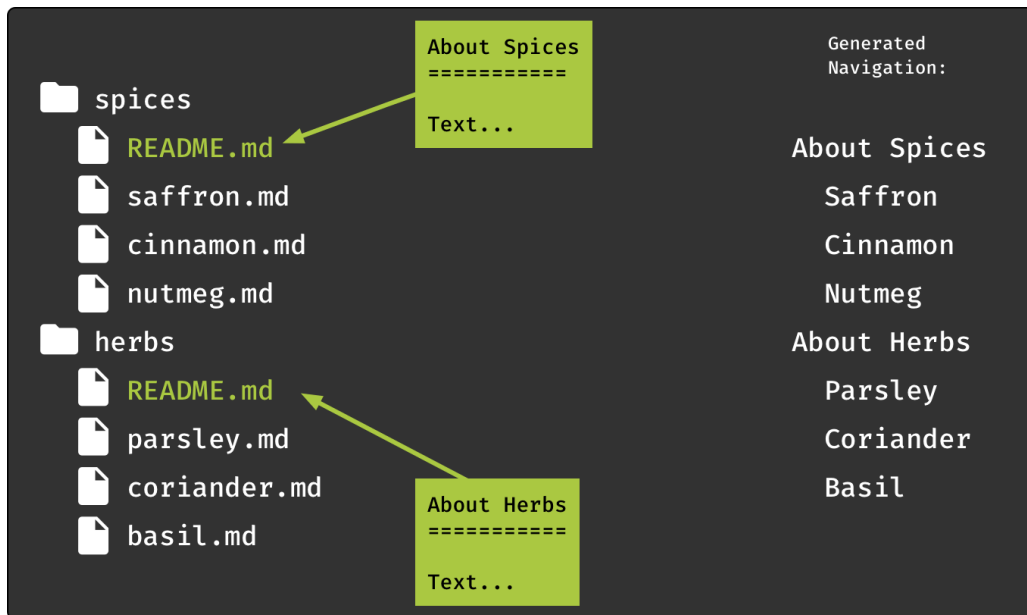
In this example each directory has a title document. The title of that document is taken from the first header. In the navigation tree on the right it is rendered one layer above the other documents in the same directory, and the title is linked to the document.

Example for a structure without title documents

```
                              laika.title = About Spices         Generated
                                                                Navigation:
    📁 spices
        📄 directory.conf                                   About Spices
        📄 saffron.md                                          Saffron
        📄 cinnamon.md                                         Cinnamon
        📄 nutmeg.md                                           Nutmeg
    📁 herbs                                                About Herbs
        📄 directory.conf                                      Parsley
        📄 parsley.md                                          Coriander
        📄 coriander.md                                        Basil
        📄 basil.md
                              laika.title = About Herbs
```
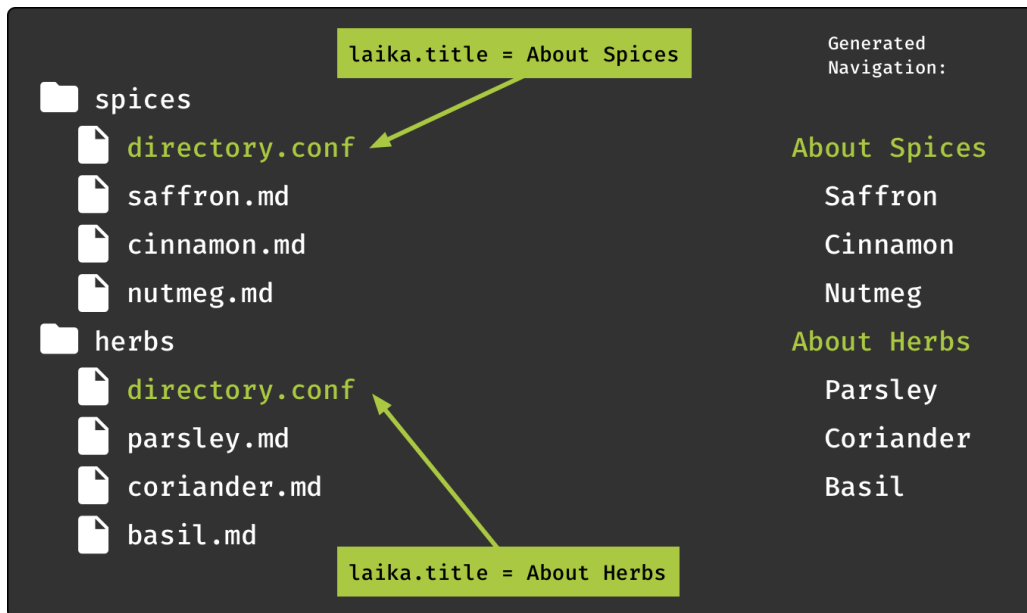
In this example there are no title documents and chapter titles are taken from the file `directory.conf` in each directory. In the navigation tree on the right these titles now only serve as separators and are not linked to any document.

Virtual Tree Abstraction

While in most scenarios a single directory to provide all input files is probably sufficient, Laika's functionality is not tied to the file system in any way, but instead builds on top of a virtual tree abstraction.

As a consequence, you can alternatively merge multiple directories into one logical tree or generate additional files programmatically and "mount" them at a specific point in the virtual tree.

When merged directories contain sub-folders with the same name, those will be merged recursively. Only files with the same name in the same folder are treated as errors.

In the example above two directories get merged into a single virtual root (`/`). The `images` directory exists in both sources, so it will be merged recursively.

The configuration mechanism for specifying multiple input directories is different for the **sbt plugin** and the **Library API**

Internally, the virtual path is represented by the `laika.ast.Path` type and relative links between them by `laika.ast.RelativePath`. These types are used frequently throughout the classes forming the document AST. All internal links are expressed with with these virtual paths and not with file system paths.

Navigation

Laika's functionality for navigation can roughly be divided into four categories:

- Features added on top of the native link syntax of supported markup formats, like validated **Validated Internal Links** or **Global Link Definitions**.

- Link directives for individual links that provide non-standard shortcuts for convenience, like **Linking by Section Headline** or **Linking to API Documentation**.

- Directives that produce larger navigation structures, like **Breadcrumbs** or **Generating Navigation Trees**.

- If you want to create your own link shortcuts, you can create **Custom Link Directives**.

Validated Internal Links

Internal Links in Laika are validated, as long as they use the source document's name (e.g. `target.md`) and not the suffix of the output document. First this ensures that the links work in all target formats, including e-books. Secondly it also means that they work on GitHub, which is convenient, when your project is hosted there.

For regular, internal links you can use the text markup's native link syntax. You can express an internal link either with a relative path:

```
Please read the [Introduction](../basics/introduction.md).
```

Or an absolute path from the root of the virtual input tree:

```
Please read the [Introduction](/basics/introduction.md).
```

The paths do not have to be valid relative or absolute paths in the file system. You can always work within the **Virtual Tree Abstraction**.

It means that a relative path like `../images/foo.jpg` can point to an `images` directory that has been merged into the input tree in the project's configuration, and does not exist relative to the source document in the file system it is linked from.

Likewise, absolute paths like `/images/foo.jpg` always refer to the virtual root of the input tree, not to the root of your file system or your website.

Disabling Validation

Transformations that contain internal links that cannot be resolved will fail in Laika's default run configuration. If you work with input files that will be generated or copied by other tools and are not known to Laika, you can explicitly disable validation for certain paths within the virtual tree:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LinkConfig(excludeFromValidation = Seq(Root / "generated")))
```

Global Link Definitions

Both supported markup formats support the use of link definitions in input files that can be referred to by link references.

Example for Markdown:

```
Please read the [Introduction][intro].
(Or in the short form, just [intro]).

[intro]: ../basics/introduction.md
```

Example for reStructuredText:

```
Please read the intro_.

.. _intro: ../basics/introduction.md
```

This is a convenient mechanism within a single input file. But if you have a larger project and use the same link definition in multiple files you would still end up with unnecessary repetition. You could use the `@:import` directive and define them in a separate markup file, but Laika has an even more convenient way to centrally define links. Simply add them to the project's configuration:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LinkConfig(targets = Seq(
    TargetDefinition("Example 1", ExternalTarget("https://example1.com/")),
    TargetDefinition("Example 2", ExternalTarget("https://example2.com/"))
  )))
```

They can then be used within text markup the same way as if they would be defined within the file:

```
Please have a look at [Example 1] or [Example 2].
```

External Links

External links can also be expressed by the text markup's native syntax, but in contrast to internal links they are not validated during a transformation.

Example for an inline link in Markdown:

```
Please read the [Specification](https://markup/specification.html).
```

Example for an indirect link via separate definition:

```
Please read the [Specification][markup-spec].

[markup-spec]: https://markup/specification.html
```

Like with internal links, **Global Link Definitions** can be used to avoid the repetition of declaring the same link target in multiple files.

Linking by Section Headline

The native linking syntax provided by the supported text markup formats can be used to directly refer to a section by its headline.

If you have the following headline in one of your documents:

```
Monkey Gone To Heaven
---------------------
```

Then you can use this title with standard link syntax:

```
Here are the lyrics for [Monkey Gone To Heaven].
```

In the example above, the section headline will also become the link title. In cases where you want to use a link text that differs from the headline text, you can also use the alternative syntax:

```
You have to listen to this [song][Monkey Gone To Heaven].
```

Resolving Ambiguity

One convenient aspect of linking by section headline is that the headline you refer to does not even have to be globally unique within your input tree. It will find the closest headline in the current scope.

For example, if you have a structure where every chapter has a page titled "Introduction", any other page within that chapter (directory) can refer to this page simply by using `[Introduction]`.

Even if the headline is not unique within the current scope, it will further disambiguate based on the level of the headline. A level 1 headline with have precedence over a level 3 headline for example.

Only duplicate headlines within the same scope and with the same level will lead to an error and cause the transformation to fail.

Linking to API Documentation

The `@:api` directive can be used to link to API documentation by providing just the fully qualified type name:

```
See @:api(scala.collection.immutable.List) for details.
```

This directive requires the base URI to be defined in the project's configuration:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LinkConfig(apiLinks = Seq(
    ApiLinks(baseUri = "https://example.com/api")
  )))
```

If you use different APIs hosted on different servers, you can associate base URIs with package prefixes, while keeping one base URI as a default for all packages that do not match any prefix:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LinkConfig(apiLinks = Seq(
    ApiLinks(baseUri = "https://example.com/api"),
    ApiLinks(baseUri = "https://somewhere-else/", packagePrefix = "com.lib42")
  )))
```

In cases where this simple mechanism is not sufficient, you can always define **Custom Link Directives**.

Linking to Source Code

The `@:source` directive can be used to link to the source code of types by providing just the fully qualified type name:

```
You can examine the source code of @:source(laika.api.Transformer) for
 inspiration.
```

It is very similar to the `@:api` directive with only a few subtle differences in its configuration. This directive requires the base URI and suffix to be defined in the project's configuration:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LinkConfig(sourceLinks = Seq(
    SourceLinks(baseUri = "https://github.com/team/project", suffix = "scala")
  )))
```

If you use link to different sources hosted on different servers, you can associate base URIs with package prefixes, while keeping one base URI as a default for all packages that do not match any prefix:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(LinkConfig(sourceLinks = Seq(
    SourceLinks(
      baseUri = "https://github.com/team/project",
      suffix = "scala"
    ),
```

```
  SourceLinks(
    baseUri = "https://github.com/elsewhere/project",
    suffix = "scala",
    packagePrefix = "com.lib42"
  )
)))
```

Generating Navigation Trees

Depending on the output formats you intend to generate, you have different options for producing tables of contents and navigation bars.

Websites

The default Helium theme provides a main navigation tree in the left sidebar and page navigation on the right out of the box.

If you include your own templates, either for overriding the Helium templates or when not using a theme at all, you can easily generate custom navigation structures with the `@:navigationTree` directive described below.

For more details on templating, see **Creating Templates**.

EPUB & PDF

Laika's E-book support automatically generates navigation elements for EPUB and PDF readers. These are not part of the templates for these output formats, therefore have to be configured separately as shown in **E-Books (EPUB & PDF)**.

You can also additionally include a table of content in the rendered pages, by using the `@:navigationTree` directive described below in the templates for EPUB and PDF.

The navigationTree Directive

This flexible directive allows to freely combine auto-generated navigation trees based on your input tree's structure with manual entries.

It inserts a `NavigationList` node into the tree - like all directives it does generate an AST element as its result and does not directly produce string output in the target format.

A navigation bar for the content of the entire project can be generated by referring to the virtual root:

```
@:navigationTree {
  entries = [ { target = "/" } ]
}
```

For generating a navigation tree for the current page with all its sections, you can refer to the current document with `#`:

```
@:navigationTree {
  entries = [ { target = "#" } ]
}
```

But you can also specify any absolute or relative path as the root of the entry:

```
@:navigationTree {
  entries = [
    { target = "../introduction" }
    { target = "../reference" }
  ]
}
```

These path references can either point to a directory or an individual document.

Limiting the Depth

By default the navigation tree will allow an arbitrary depth and not only reflect the hierarchy of directories and document and their titles, but also include the individual sections within the documents.

If you want to restrict the tree to a particular navigation depth, you can do so with the `defaultDepth` attribute on the root level, or the `depth` attribute on each individual entry:

```
@:navigationTree {
  entries = [ { target = "/", depth = 3 } ]
}
```

Excluding Document Sections

If you want to generate a pure document-based navigation, which is quite common for the main navigation side bar, you can also generally disable section links with:

```
@:navigationTree {
  entries = [ { target = "/", excludeSections = true } ]
}
```

This way it will render documents as the leaf nodes and ignore their sections, even if the maximal depth has not been reached yet. This attribute can be used on the root level where it applies to all entries, or for each individual entry as shown in the example above.

Excluding the Root Node

By default each entry listed will be represented by its root node with its title, with all children nested one level below. Alternatively, when `excludeRoot` is set, the root can be skipped and its children inserted into the final structure directly.

Specifying Additional Styles

By default, all navigation items get the styles `nav` and `levelN` applied, with N being the level number. For HTML output these will then be rendered as class attributes.

If you generate different kinds of navigation trees and want additional class attributes for styling them differently, you can set the `itemStyles` attribute:

```
@:navigationTree {
  itemStyles = [ "page-nav" ]
  entries = [ { target = "/", depth = 3 } ]
}
```

Adding Manual Nodes

Finally you can also include hard-coded link targets in addition to the auto-generated nodes:

```
@:navigationTree {
  entries = [
    { target = "/", depth = 3, excludeRoot = true }
    { target = "http://twitter.com/me-me-me", title = "Twitter" }
  ]
}
```

In this example an auto-generated navigation tree with depth 3 will be inserted, but an additional element with an external link target will be added as the final node.

Breadcrumbs

The classic `@:breadcrumb` component generates a simple, flat list of links reflecting the directory structure from the root to the current document:

```
Home / Spices / Cinnamon
```

This component works best in a directory structure that contains **Title Documents**, as otherwise the segments which represent a directory level would have nowhere to link to and would render as plain text.

Auto-Numbering

Laika supports auto-numbering of documents or sections or both. The term sections in Laika refers to structure obtained from the headlines within documents.

If you enable both the section numbers will be added to the document number. E.g. in the document with the number `2.1` the number for the first section will be `2.1.1`.

Auto-numbering can be switched on per configuration:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(AutonumberConfig(maxDepth = 3))
```

The configuration above will number both documents and sections within documents, but stop after the third level. The numbers will be added to the headers of the sections and also appear in tables of contents.
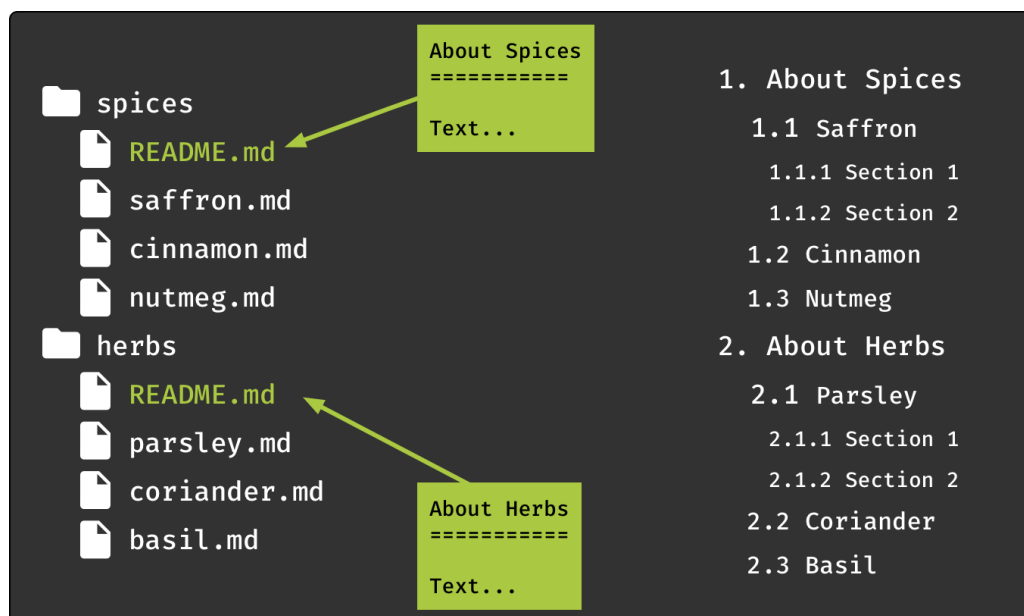
The default setting for Laika has auto-numbering switched off completely.

Custom Link Directives

The directives presented in this chapter are only a default selection of navigation features that hopefully already cover most common scenarios.

If you have very specific requirements you can always fall back to implementing your own directives. You can examine the implementation of Laika's default link directives as an example.

For details see **Implementing Directives**.

Theme Settings

If you use the sbt plugin or the parsers, renderers or transformers from the `laika-io` module, each transformation is based on a theme that provides a set of default templates and styles for all output formats.

If you use a pure transformer from the `laika-core` module, there is no theme support as these transformers only process a single string as input and output.

This chapter documents Laika's only built-in theme, called `Helium`, and its configuration options.

The Helium Theme

The name of Laika's default theme reflects its lightweight nature as it is not based on any CSS or JavaScript libraries like Bootstrap, but instead only comes with a minimal set of hand-crafted CSS and JavaScript.

Helium includes default styles for web sites, EPUB and PDF documents, so you can generate a documentation site complete with a download page for e-book formats in one go.

The generated site is responsive and designed to work well on smaller devices.

Without any explicit theme configuration, Laika transformers run with the Helium theme and all its defaults, which is equivalent to the following configuration:

```
laikaTheme := Helium.defaults.build
```

If you are fine with your site and e-books looking exactly like Laika's documentation, including its choices for colors and fonts, you can skip the theme configuration step.

Otherwise you have the following options:

- To tweak just some aspects of the look and feel like the color schemes and fonts and to add custom links, metadata and custom styles to the default theme, follow the instructions in this chapter below.

- If you want full control and design everything from scratch, you can alternatively install `Theme.empty` instead of Helium's defaults and then add all templates and CSS files to your input directories. See **Creating Templates** for details.

- If you want full control, but re-use your design across multiple projects or publish it as a 3rd-party theme, see **Creating Themes** for details.

- Hopefully at some point there is also a selection of 3rd-party themes available.

Helium Configuration API

Helium can be fully configured with its Scala API and does not require any configuration files.

The entry point for all configuration steps is always `Helium.defaults` to ensure there are sensible defaults in place for all the options you omit in your configuration.

For each configuration step you need to choose one of the four selectors, either `all` to specify options for all three output formats or `site`, `epub` or `pdf` to select a single format. Not all options are available for all formats, but the IDE's context help and the documentation below can guide you.

In the minimal example below we only specify some metadata for all formats as well as the navigation depth for EPUB and PDF:

```
val theme = Helium.defaults
  .all.metadata(
    title = Some("Project Name"),
    language = Some("de"),
  )
  .epub.navigationDepth(4)
  .pdf.navigationDepth(4)
  .build
```

Laika also provides convenient constructors for some of the data types used frequently in its theme API. You can import `laika.theme.Color._` for specifying colors with `hex("ffaaff")` or `rgb(255, 0, 0)` and `laika.ast.LengthUnit._` for specifying sizes with `px(12)` or `pt(9)` or other available units. All configuration examples in this chapter assume these imports are in place.

The final call to `build` produces a `ThemeProvider` that can be passed to Laika's transformers or the `laikaTheme` sbt setting:

```
laikaTheme := theme
```

Fonts

By default Laika includes three fonts in its published artifact so that they are ready to be embedded into EPUB or PDF documents. All these fonts are licensed under the Open Font License:

- `Lato` (for headlines and body text): **https://fonts.google.com/specimen/Lato#about**

- `Fira Code` (for code blocks and inline code): **https://github.com/tonsky/FiraCode**

- `IcoFont` (for Laika's small set of icons): **https://icofont.com/icons**

You can override these defaults by defining your own set of fonts.

This is an excerpt of Laika's default configuration as an example:

```
val latoURL = "http://fonts.googleapis.com/css?family=Lato:400,700"
val firaURL =
  "https://cdn.jsdelivr.net/gh/tonsky/FiraCode@1.207/distr/fira_code.css"

Helium.defaults.all.fontResources(
```

```
  FontDefinition(
    Font
      .embedResource(fontPath + "Lato/Lato-Regular.ttf")
      .webCSS(latoURL),
    "Lato", FontWeight.Normal, FontStyle.Normal
  ),
  FontDefinition(
    Font
      .embedResource(fontPath + "Lato/Lato-Bold.ttf"),
    "Lato", FontWeight.Bold, FontStyle.Normal
  ),
  FontDefinition(
    Font
      .embedResource(fontPath + "FiraCode/FiraCode-Medium.otf")
      .webCSS(firaURL),
    "Fira Code", FontWeight.Normal, FontStyle.Normal
  ),
  FontDefinition(
    Font
      .embedResource(fontPath + "icofont/icofont.ttf")
      .webCSS("../icons/icofont.min.css"),
    "IcoFont", FontWeight.Normal, FontStyle.Normal
  )
)
```

The `embedResource` method picks the font to be embedded from the resource directory inside the jar, use `emedFile` to pick a font from the file system. Font embedding will be performed for EPUB and PDF output For the site you can use the `webCSS` method to link to the CSS containing the `font-face` definitions. It will automatically be included in the `<head>` sections of the generated HTML without the need to adjust the default templates.

Note that the use of the `fontResources` method *replaces* the defaults, it does not add them. If you want to use some of the built-in fonts you have to redefine them in this step. This ensures that no unused font resources will be embedded.

Additionally you can define the font families and font sizes for the site and e-books:

```
Helium.defaults
  .all.fontFamilies(
    body = "MyBodyFont",
    headlines = "MyHeadlineFont",
    code = "MyCodeFont"
  )
  .site.fontSizes(
    body = px(15),
    code = px(14),
    title = px(34),
    header2 = px(28),
    header3 = px(20),
    header4 = px(15),
    small = px(12)
```

```
  )
  .pdf.fontSizes(
    body = pt(10),
    code = pt(9),
    title = pt(24),
    header2 = pt(14),
    header3 = pt(12),
    header4 = pt(11),
    small = pt(8)
  )
```

The example above specifies the three font families with the `all` selector, defining them for all output formats. Make sure the names match font resources defined in the previous step so that they are available for embedding.

The font sizes on the other hand will often be defined per format, the example above shows the definitions for the site and PDF output.

Colors

Helium let's you define two different color sets: The theme colors which are used for headlines, links, navigation and backgrounds as well as the syntax coloring scheme to be used for all code blocks.

Theme Colors

The following example overrides Laika's theme colors for all output formats:

```
Helium.defaults
  .all.themeColors(
    primary = hex("007c99"),
    primaryDark = hex("931813"),
    primaryLight = hex("095269"),
    primaryMedium = Color.hex("a7d4de"),
    primaryLight = Color.hex("ebf6f7"),
    text = Color.hex("5f5f5f")
  )
```

The default theme colors, which are used for the documentation you are reading right now, are as follows:

primary	secondary	primaryDark	primaryMedium	primaryLight
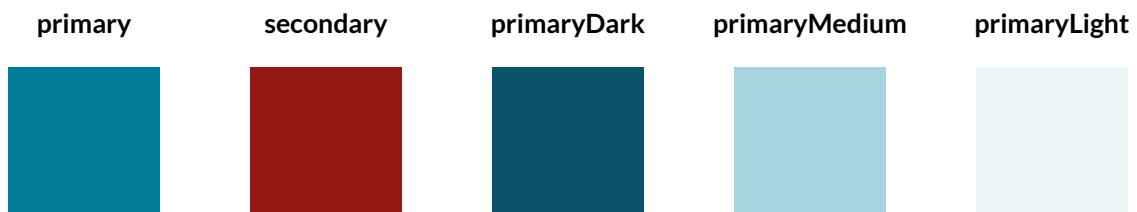
- `primary` is used for highlights and the text of navigation panes and inline code spans.

- `secondary` is used for links and headlines.

- `primaryDark` is usually a darker shade of the primary color, currently only used for the gradient on the landing page.

- `primaryMedium` is usually a shade slightly darker than `primaryLight`, used for borders and text on dark background.

- `primaryLight` is usually a lighter shade of the primary color, used as background color.

- `text` is solely used as the color of the body text.

In terms of getting the right contrast, the color combinations must support the following:

- Text in `primary` color must be readable on `primaryLight` background.

- Text in `primaryMedium` color must be readable on `primary` background (this combination is only used on the landing page).

Syntax Highlighting

These settings are only relevant if you use Laika's built-in highlighters, in case you prefer to use a 3rd-party library like `highlight.js` you need to use their styling mechanism.

Laika's own syntax color scheme is built around sets of 10 colors, of which 5 are usually a set of grayish colors from light to dark with almost no saturation and a set of 5 colors which are supposed to be positioned around the color wheel.

The defaults for Laika's dark theme (as used on this site) are:

base-1	base-2	base-3	base-4	base-5

wheel-1	wheel-2	wheel-3	wheel-4	wheel-5
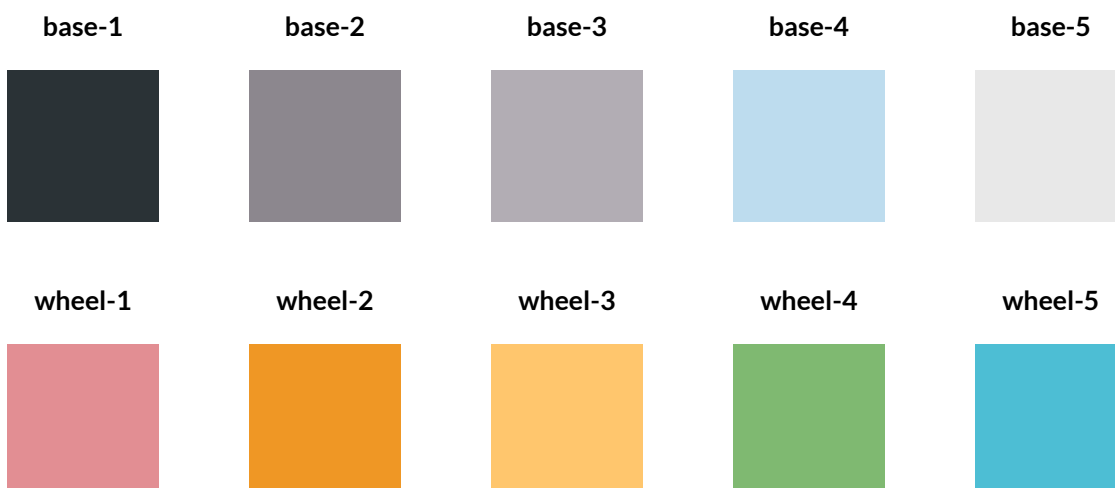
For the light scheme (used by default in EPUB and PDF output) they are:

base-1	base-2	base-3	base-4	base-5

wheel-1	wheel-2	wheel-3	wheel-4	wheel-5
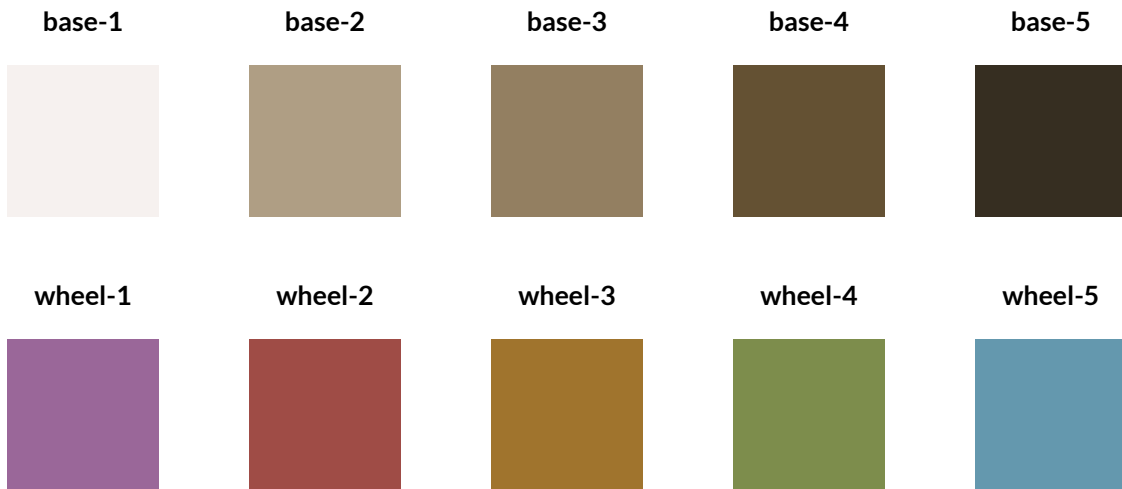
They can be overridden per format or for all at once:

```
Helium.defaults
  .all.syntaxHighlightingColors(
    base = ColorQuintet(
      hex("2a3236"), hex("8c878e"), hex("b2adb4"),
      hex("bddcee"), hex("e8e8e8")
    ),
    wheel = ColorQuintet(
      hex("e28e93"), hex("ef9725"), hex("ffc66d"),
      hex("7fb971"), hex("4dbed4")
    )
  )
```

For dark themes the base colors have to go from dark to light, for light themes the other way round.

The base colors are used as background color (the first), for comments, punctuation, identifiers and uncategorized spans, the wheel colors are used for the more significant types of code spans. Amongst others these are substitution references, annotations (1st color), keywords, escape sequences (2nd), attribute names, declaration names (3rd), all types of literals (4th), type names, tag names (5th), and several categories for other types of syntax, e.g. in text markup.

Layout

The configuration API allows to control a few aspects of the layout, like columns widths and line heights, that are the most like candidates for frequent adjustments.

These are the options for the site:

```
Helium.defaults
  .site.layout(
    contentWidth = px(860),
    navigationWidth = px(275),
    defaultBlockSpacing = px(10),
    defaultLineHeight = 1.5,
    anchorPlacement = AnchorPlacement.Left
  )
```

The example shows the values for Helium's defaults. `contentWidth` is the maximum width of the main column, `navigationWidth` the width of the left navigation pane, `defaultBlockSpacing` the space between block elements in the main column and `defaultLineHeight` the line height of normal paragraphs.

`anchorPlacement` configures the little anchors for section headers that appear on mouse-over and allow to copy the URL. The default is `Left`, other options are `Right` or `None`.

If you need to tweak the layout beyond these options you can always include custom CSS files in your input directory that override additional styles.

For PDF output you can also define the page size and margins:

```
Helium.defaults
  .pdf.layout(
    pageWidth = cm(21),
    pageHeight = cm(29.7),
    marginTop = cm(1),
    marginRight = cm(2.5),
    marginBottom = cm(1),
    marginLeft = cm(2.5),
    defaultBlockSpacing = mm(3),
    defaultLineHeight = 1.5,
    keepTogetherDecoratedLines = 12
  )
```

Again, the example shows the defaults. The properties for the page dimensions and margins should be self-explanatory, `defaultBlockSpacing` and `defaultLineHeight` are analogous to the site settings. Finally, `keepTogetherDecoratedLines` controls the number of lines for decorated blocks like code examples or callouts that should always be kept on the same page. With the above setting of 12 only blocks with more than 12 lines are allowed to be split across multiple pages.

Metadata

Helium allows to define a small set of metadata that describes the output. In the generated site it will be used to populate the `<head>` section of the HTML output, for EPUB and PDF it will be used to embed the information into the generated files in a way that the respective readers understand.

Like many other options, you can specify metadata either with the `all` selector for all formats, or define them separately with the `epub`, `pdf` or `site` selectors.

```
Helium.defaults
  .pdf.metadata(
    title = Some("Project Name"),
    description = Some("Fancy Hyperbole goes here"),
    identifier = Some("XSD-9876-XVT"),
    authors = Seq("Maria Green", "Helena Brown"),
    language = Some("de"),
    date = Some(Instant.now),
    version = Some("2.3.4")
  )
```

When using the sbt plugin the `title`, `description` and `version` metadata will be pre-populated by the standard sbt settings `name`, `description` and `version` respectively. When using the library API no medata will be defined by default. It is recommended to always define the language and title as the minimum set of metadata.

Navigation, Links & Favicons

This section describes the navigation options specific to the Helium theme. There are several other navigation features which are supported independent of the theme in use, like the option to centralize all definitions of external links. They are described in a dedicated **Navigation** chapter.

Top Navigation Bar

You can place a logo serving as home button into the middle of the top navigation bar, and a row of buttons or icons with links at the right corner. The left corner of the top bar is always reserved for the button to open or close the navigation.

```
Helium.defaults.site
  .topNavigationBar(
    logo = Some(Logo.internal(
      path = Root / "image.png",
      alt = Some("Homepage"),
      title = Some("Home")
    )),
    links = Seq(
      IconLink.internal(Root / "doc-2.md", HeliumIcon.download),
      ButtonLink.external("http://somewhere.com/", "Text Link"),
      ButtonLink.external("http://somewhere.com/", "Button Link")
    ))
```

The logo can be an external or internal link, in case of the latter, it is always a path from the perspective of Laika's virtual root, not a file system path. For the links to the right navigation bar you can choose between an `IconLink` with optional text, a `ButtonLink` with an optional icon, or a plain text link. Internal targets are again within the virtual path and will be validated.

With the default Helium settings the three link types from our code example render as shown below:

Table of Contents

All three formats support an additional table of contents. This does not refer to the main navigation menu (the left pane of the Helium site and the respective reader's navigation bars in case of EPUB or PDF) which Helium will always produce. It is an optional, separate page that can be included in e-books or the site. For PDF format this is essential, as the reader's navigation would be lost if the user prints the PDF.

```
Helium.defaults
  .site.tableOfContent(title = "Contents", depth = 3)
  .pdf.tableOfContent(title = "Contents", depth = 4)
  .epub.tableOfContent(title = "Contents", depth = 4)
```

For all formats you can specify the title of the page and the navigation depth. The depth value includes all layers of the hierarchy, directories, documents and sections within documents.

In e-books the table of content will be inserted right after the (optional) cover image, for the site it will become a link at the top of the left navigation pane.

Finally, the navigation depth for the reader's navigation for EPUB and PDF can also be controlled:

```
Helium.defaults
  .epub.navigationDepth(4)
  .pdf.navigationDepth(4)
```

> ℹ️
>
> The default for EPUB is just 2 levels as some readers like iBooks mess with the hierarchy of navigation items when using more than 2 levels. If you increase this setting make sure you verify it's looking good in the targeted readers.

Links to Markup Sources

A Helium site contains a page navigation pane on the right side on bigger screens. At the bottom of that pane a link to the source of the current page will be inserted if you provide the base URL to link to:

```
Helium.defaults
  .site.markupEditLinks(
    text = "Source for this Page",
    baseURL = "https://github.com/my/project"
  )
```

Favicons

You can specify one or more favicons for your site:

```
Helium.defaults
  .site.favIcons(
    Favicon.internal(Root / "favicon32x32.png", sizes = "32x32"),
    Favicon.internal(Root / "favicon64x64.png", sizes = "64x64")
  )
```

Download Page

A page offering generated EPUB and PDF files with the same content and structure as the site for download can be added via Helium configuration:

```
Helium.defaults
  .site.downloadPage(
    title = "Documentation Downloads",
    description = Some("Optional Text Below Title"),
    downloadPath = Root / "downloads",
    includeEPUB = true,
    includePDF = true
  )
```

The last three properties in the example above are equivalent to the defaults and can be omitted.

If you configure **Cover Images for E-books** they will be used to display thumbnails above the download links.

Website Landing Page

By default the generated site does not have a dedicated landing page. The site's entry point will simply be the `README.md` or `README.rst` document (if present), rendered with the default template, meaning with the top and left navigation bars. If you don't mind that users arrive straight at the content pages, you can stick with the defaults.

Alternatively Helium offers a dedicated landing page tailored for software documentation sites. This is the full set of content options available:

```
Helium.defaults
  .site.landingPage(
    logo = Some(Image()),
    title = Some("Project Name"),
    subtitle = Some("Fancy Hyperbole Goes Here"),
    latestReleases = Seq(
      ReleaseInfo("Latest Stable Release", "2.3.5"),
      ReleaseInfo("Latest Milestone Release", "2.4.0-M2")
    ),
    license = Some("MIT"),
```

```
    documentationLinks = Seq(
      TextLink.internal(Root / "doc-1.md", "Doc 1"),
      TextLink.internal(Root / "doc-2.md", "Doc 2")
    ),
    projectLinks = Seq(
      IconLink.internal(Root / "doc-2.md", HeliumIcon.demo),
      ButtonLink.external("http://somewhere.com/", "Somewhere")
    ),
    teasers = Seq(
      Teaser("Teaser 1", "Description 1"),
      Teaser("Teaser 2", "Description 2"),
      Teaser("Teaser 3", "Description 3")
    )
  )
)
```

Every single content item shown above is optional, but of course the page would look strange if you omit them all.

The diagram below shows the positions of these items on the page:
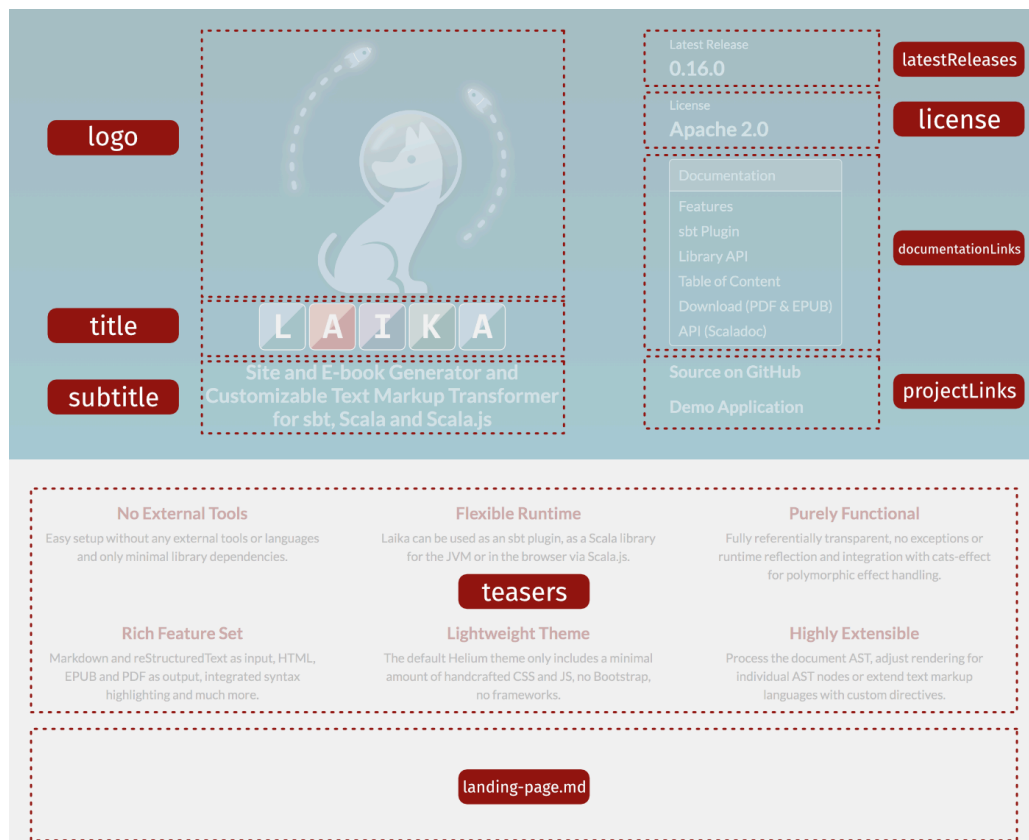
The left side of the header introduces the project, ideally you would choose at least one of the three options (logo, title and subtitle). In the case of Laika's site for example, the title is omitted as the project name is already part of the logo.

On the right side, the latest release info usually points to one or two releases, the latter if there is also a milestone available. The panel for documentation links can be any links right into the content of your site, like Getting Started pages, tables of contents, and hopefully also a link to the API documentation.

The project links below can be any set of additional links, like to GitHub, Twitter or your chat. Like with the top navigation bar of the main page, you can choose between an `IconLink` with optional text, a `ButtonLink` with an optional icon, or a plain `TextLink`. Internal targets are again within the virtual path and will be validated.

> ℹ
>
> The right side of the header is the only section of the landing page containing links. This is a deliberate departure from the often confusing entry points of other software documentations sites where relevant links are scattered all over the place.

Finally a set of teasers which are quite common on landing page can be defined for the white area below the header.

Additionally or alternatively you can also add a regular markup document called `landing-page.<suffix>` to one of your input directories and its content will be inserted at the bottom of this page. This is in case you want to stick with the traditional approach of some sites, which give you 'Getting Started' style content right on the start page. It can also be used to list adopters, provide a feature overview or links to presentations or videos.

Cover Images for E-books

You can include cover images for EPUB and PDF files. They will also be used to display little thumbnails on the download page.

```
Helium.defaults
  .epub.coverImages(CoverImage(Root / "cover.png"))
  .pdf.coverImages(CoverImage(Root / "cover.png"))
```

See **Supported Document Types** for a list of supported image formats.

Auto-Linking CSS & JS Files

If all the configuration options shown above don't give you enough flexibility, you can always include additional CSS and JavaScript files to tweak the site's look & feel.

You can examine the CSS generated by Helium and override individual style declarations by placing them anywhere in a CSS file inside one of your input directories. They will always be included in a way that they have higher precedence than the theme's styles.

If you want to prevent Helium from scanning all input directories for CSS files, you can alternatively specify one or more directories explicitly:

```
Helium.defaults
  .site.autoLinkCSS(Root / "my-css")
  .site.autoLinkJS(Root / "my-js")
```

Like everything in Laika, the paths are virtual and not file system paths, so they must point to somewhere within the configured inputs. .

> **ℹ**
>
> The use of JavaScript in EPUB templates is not yet supported, but planned for version 0.17.
>
> The reason is that EPUB requires that HTML pages containing scripts are marked as such in the metadata files that list the content of the EPUB, which means that Laika would first need to introduce configuration support for authors to specify which templates contain scripts.

Custom Templates

In cases where even custom CSS files do not give you the level of control you need, the final step (apart from creating your own theme), would be to create your own default templates.

As a starting point you can copy and modify Helium's default templates, which you can find **here**. Or you can start from scratch and only use the existing templates as an inspiration.

Any default templates placed into the user input directories will override the corresponding templates with the same path from a theme. This way you can always customize just one of the templates, while keeping the rest as provided by the theme.

For an introduction to the template syntax, template file names and more details see **Creating Templates**.

E-Books (EPUB & PDF)

Laika puts equal emphasis on the generation of e-books as on website generation.

Like all functionality, e-book generation does not require installation of external tools. The EPUB format is supported by Laika's own Scala implementation, whereas PDF support is partially based on **Apache FOP**.

The latter is not based on LaTeX like many other PDF renderers, but uses XSL-FO as an interim format. The generation of this interim format happens in Laika's own renderers, only the final transformation step to the binary PDF format is delegated to Apache FOP.

Generating E-Books

If you are using the sbt plugin you can use several of its task for generating e-books:

- `laikaEPUB` or `laikaPDF` for transforming a directory of input files to a single output format

- `laikaGenerate epub pdf <other formats>` for transforming a directory of input files to EPUB and PDF and other output formats with a single parse operation

- `laikaSite` for generating a site that also contains PDF and/or EPUB files for download when setting the `laikaIncludeEPUB` and/or `laikaIncludePDF` settings to true.

See **sbt Plugin** for more details.

When using the library API, the `EPUB` and `PDF` renderers can be passed to the `Transformer` or `Renderer` APIs:

```scala
val transformer = Transformer
  .from(Markdown)
  .to(EPUB)
  .using(GitHubFlavor)
  .io(blocker)
  .parallel[IO]
  .build

transformer.use {
  _.fromDirectory("src")
   .toFile("hello.epub")
   .transform
}
```

See **Library API** for more details on these APIs and ensure you have the necessary dependencies in place - the `laika-pdf` module for PDF or the `laika-io` module for EPUB - see **Dependencies**.

Directory Structure

An e-book generated by Laika is always a single, binary file, even when the input is an entire directory of markup files. But the directory structure will be reflected in the navigation elements inside the e-book.

Markup files from the input directory will form the content of the e-book while static files like images and fonts will be embedded into the output when they are referenced from within one or more markup files.

For EPUB specifically it is important to consider what file size to choose for your content. The directory structure inside the generated EPUB container (which is essentially a glorified ZIP) will mirror exactly the structure of the input directory, apart from the additional metadata files it needs to generate. Therefore it is best to avoid very large markup files as they might slow down the experience in the e-book reader, and instead distribute content over a higher number of smaller input files.

See **Document Types** for general info about the various types of input files Laika supports, and **Supported Document Types** in this chapter for additional info specific to EPUB and PDF.

Configuration

The most convenient way to configure aspects like metadata, look & feel and navigation for your e-books is to use the built-in default theme called Helium and its settings.

There is a dedicated **Theme Settings** chapter, so we'll just link the most relevant aspects that can be configured for e-books from here.

- The **Fonts** to embed in EPUB and PDF files in case you do not want to use Helium's default fonts.

- The **Colors** for the main theme and for syntax highlighting.

- Several aspects of the theme's **Layout**, like line height, block spacing or PDF page sizes.

- **Metadata** (title, authors, description, language, etc.) to include in the generated e-books in a way that reader software can expose.

- Configure **Cover Images for E-books** or **Auto-Linking CSS & JS Files**.

Book Navigation

Laika supports a directory structure with sub-directories of any depth. Markup files from the input directory will form the content of the e-book, linearized in depth-first traversal and using your configured **Navigation Order**.

Laika will generate navigation elements compatible with e-book readers, as shown in the images below:

PDF Navigation in Preview for Mac:
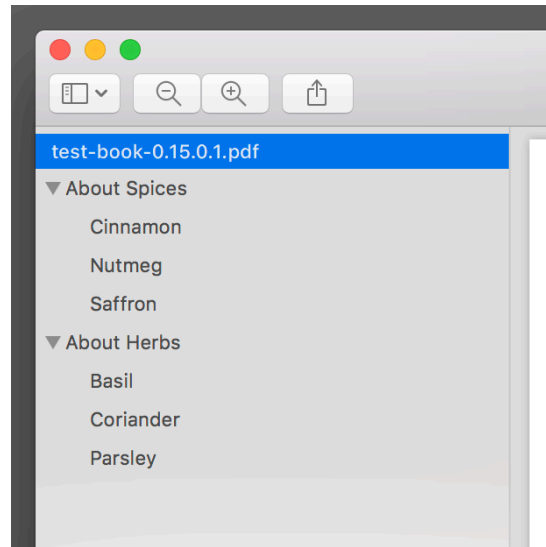
EPUB Navigation in iBooks:
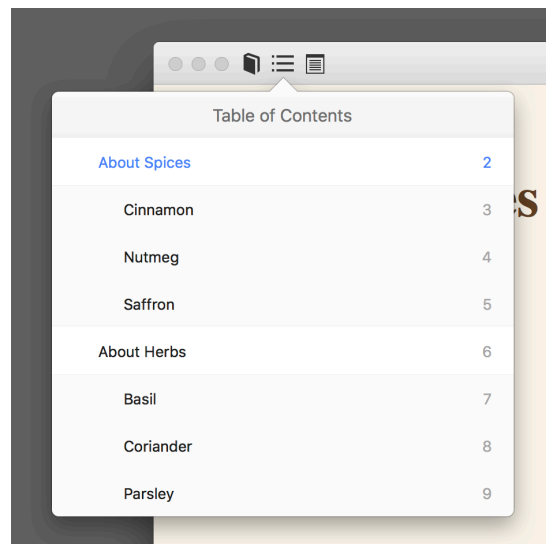
The navigation depth can be configured with the Helium API:

```
Helium.defaults
  .epub.navigationDepth(4)
  .pdf.navigationDepth(4)
```

> ℹ️
>
> The default for EPUB is just 2 levels as some readers like iBooks mess with the hierarchy of navigation items when using more than 2 levels. If you increase this setting make sure you verify it's looking good in the targeted readers.

Note that this affects the navigation structure that will be generated for the navigation tools of the respective EPUB or PDF readers.

You might additionally want to insert a table of content into the page flow, right after the cover image, as this would be the only navigation structure that would be available when printing the document. See **Table of Contents** how to configure such a structure which can also have a different navigation depth than the tree generated for the readers.

Supported Document Types

You can also place images, fonts and other supported file types into the input directory. Laika will embed these files in the generated EPUB container and/or PDF file.

The supported file types / suffixes are:

For EPUB

- Images: `jpg`, `jpeg`, `gif`, `png`, `svg`

- Audio: `mp3`, `mp4`

- HTML: `html`, `xhtml`

- JavaScript: `js`

- CSS: `css`

- Fonts: `woff2`, `woff`, `ttf`, `otf`

For PDF

- Images: `jpg`, `jpeg`, `gif`, `png`, `svg`, `bmp`, `tiff`

- Fonts: `pfb` (Type 1), `ttf`, `otf`

CSS for EPUB

Since content files for EPUB are standard XHTML files (apart from optional EPUB-specific attributes), you can style your e-books with standard CSS.

It is sufficient to simply place all CSS into the input directory, alongside the text markup and other file types. References to these CSS files will be automatically added to the header section of all generated HTML files.

To enable a distinction between EPUB and web site generation in case you want to produce both with the same inputs, Laika expects the following suffixes for CSS files:

- Files ending with `.epub.css` will only be linked in HTML files for EPUB, not for the site

- File ending with `.shared.css` will be linked for both

- All other files ending with `.css` will only be used for web site content

When referencing images or fonts from your CSS files, you can use relative paths, as the directory layout will be retained inside the EPUB container.

CSS for PDF

Laika offers the unusual, but convenient feature of CSS styling for PDF. It allows for customization in a syntax familiar to most users.

The CSS files need to be placed into the root directory of your input tree with a name in the format `<name>.fo.css`.

However, as PDF is a page-based format and in Laika's case expects `XSL-FO` as an interim format, there are a few subtle differences:

- All CSS attributes must also be valid XSL-FO attributes. There is quite a big overlap, so you can use the familiar `font-family`, `font-weight`, `font-size` attributes as well as most of the border, padding and margin attributes. For an overview over the available attributes you can refer to the **Formatting Properties** chapter in the XSL-FO specification.

- While id selectors and class selectors function like in web CSS, the type selectors are quite different. Since there is no interim HTML result, the types you can refer to are not HTML tag names like `div`, but the class names of the Laika AST nodes, e.g. `Header` or `Paragraph`. See **The Document AST** for details.

- Some selector types of the CSS specification are not supported as most of them do not add much value in the context of a Laika document tree:

  - Pseudo-classes like `:hover`

  - Attribute selectors like `[attribute~=value]` (since Laika nodes do not have many properties)

  - `Element1+Element2` or `Element1~Element2` for selecting based on sibling elements

An example for styling a level-2 header:

```
Header.level2 {
  font-family: sans-serif;
  font-weight: bold;
  font-size: 14pt;
}
```

The `Header` type selector refers to the name of the AST node Laika produces for headers. The `level2` class is a style that gets rendered for each header node to be able to style levels differently

Only a few of Laika's nodes get rendered with class attributes. You can examine HTML output to get an overview over rendered attributes as they are largely identical to those rendered for EPUB and PDF.

Likewise, out of the box only a subset of nodes get rendered with an auto-generated id: headers, footnotes or citations.

Customized Output

In cases where styling with CSS alone is not sufficient, there are additional, lower-level hooks to customize the rendered output.

Templates

You can place custom default templates into the root directory of the input tree, named `default.template.epub.xhtml` for EPUB and `default.template.fo` for PDF.

For EPUB the template will be applied to each rendered file individually, for PDF the generated XSL-FO will first be concatenated and then the template will be applied to the single final FO document.

Customizing the PDF template would require knowledge of `XSL-FO`, but is hopefully rarely ever necessary as PDFs can be styled by Laika's **CSS for PDF** feature.

See **Creating Templates** for general info about Laika's template engine.

Overriding Renderers

Apart from adjusting the surrounding template the AST nodes will be rendered into, you can also customize how each individual AST node itself is rendered.

This is a general customization hook in the library and not different from overriding renderers for site output. See **Overriding Renderers** for more details.

Syntax Highlighting

Laika has its own built-in syntax highlighters (based on its parser combinators). Not relying on external tools for this task has several advantages:

- The highlighting is not tied to HTML output and works in the same way for EPUB or PDF output.

- The analyzed code is part of the document AST and can be processed and transformed like other AST nodes before rendering.

This manual itself is a showcase for this functionality. All code samples shown are highlighted by Laika's own syntax support.

Configuration

Laika's syntax highlighting is not enabled by default, so that you can still choose other existing solutions if you prefer.

When using the sbt plugin it can be added to the `laikaExtensions` setting:

```
laikaExtensions ++= Seq(GitHubFlavor, SyntaxHighlighting)
```

Note that you also have to add the `GitHubFlavor` extension so that you get the support for fenced code blocks. When using reStructuredText input only the `SyntaxHighlighting` extension is needed.

When using the Library API highlighting can be activated like all other extension bundles:

```
val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor, SyntaxHighlighting)
  .build
```

Supported Languages

Laika currently supports the following languages and formats:

- Scala

- Dotty

- Java

- Python

- JavaScript / JSX

- TypeScript / TSX

- HTML

- CSS

- XML

- JSON

- HOCON

- SQL

- EBNF

- Laika Extensions

The last highlighter can be used when writing documentation about Laika itself: it includes highlighters for Laika's directives, substitution references and configuration headers.

More languages will be added over time (contributions are also welcome of course).

Styling

The default Helium theme contains color sets for HTML, EPUB and PDF that can be overridden via the configuration API as shown in **Theme Settings / Syntax Highlighting**.

If you do not only want to specify the color scheme, but also how code categories are grouped into the color set, you can alternatively add custom CSS. The classes generated for code spans are:

- `comment`

- `keyword`

- `boolean-literal`

- `number-literal`

- `string-literal`

- `char-literal`

- `symbol-literal`

- `regex-literal`

- `literal-value`

- `escape-sequence`

- `substitution`

- `type-name`

- `attribute-name`

- `identifier`

- `tag-name`

- `tag-punctuation`

- `xml-dtd-tag-name`

- `xml-processing-instruction`

- `xml-cdata`

- `markup-fence`

- `markup-headline`

- `markup-emphasized`

- `markup-quote`

- `markup-link-text`

- `markup-link-target`

Integrating External Tools

If you want to use the available highlighters, but also need to use languages not currently supported out of the box, you can integrate external highlighters just for those languages.

The only thing you need to ensure is that those 3rd-party tools do not try to mess with the code blocks that have already been processed by Laika.

For `highlight.js` Laika already provides this kind of blocker, in that it adds the `nohighligting` class to all code blocks it has already analyzed itself.

If you are using different tools you might need to customize the rendering of the `CodeBlock` AST element. For details see **Overriding Renderers**.

Adding Your Own Highlighters

Creating your own highlighter can potentially be quite easy, in cases where the syntax can be defined mostly by combining Laika's building blocks for parsing string and number literals, identifiers and keywords. For a fairly simple highlighter implementation as an example you can examine the **ScalaSyntax** source as a starting point.

Once you have implemented and tested your highlighter you can add it to the built-in ones like this:

```
laikaExtensions ++= Seq(
  GitHubFlavor,
  SyntaxHighlighting.withSyntax(MyHighlighter)
)
```

See **Adding Syntax Highlighters** for details.

Overview

Laika's Documentation comes with separate sections for "Customizing" and "Extending" Laika. While the line between the two is naturally quite blurry, the distinction is mostly meant to be between these two scenarios:

- Customizing Laika refers to the things you most likely want to tweak and adjust for a specific project.

- Extending Laika refers to the kind of re-usable extensions you might want to write as an in-house library or a 3rd-party open source extension.

This section deals with the former scenario.

Transformation Phases

In cases where you just want to tweak some minor details about how your site or e-book renders, you can use one of Laika's hooks into the various phases of a transformation.

For a better understanding of the customization options it is good to a have a rough idea about how a transformation in Laika is performed. It can be divided into 4 phases:

1) The parsing step. Text markup and template documents get parsed and translated into an internal AST. The AST is a generic abstraction of the document's structure and is not tied to any specific semantics of a particular input or output format.

2) The AST transformation. The original AST is only what each parser for the corresponding block or inline element can process locally, without access to other nodes or even other documents. One of the advantages of this design, apart from separation of concerns, is that parsers can run in parallel. As a consequence nodes like internal references or auto-numbered footnotes require further processing with access to a `DocumentCursor` that allows to access content from anywhere in the input tree.

3) Applying templates to markup documents. Since both are just AST structures, this step is merely a second AST transformation. The AST representing the markup document will be inserted into the node of the template AST that holds a reference to the content.

4) Rendering. As the last step the final AST obtained from the two previous transformation steps will get rendered to one or more output format. This is the only step specific to a particular output format, meaning the same AST structure obtained in 3) will get used as the input for the renderers of all formats.

Customization Options

The hooks that are the most likely candidates of helping you with smaller tweaks to a transformation are the following:

- **Creating Templates**: Hooks into phase 3 described above. A template lets you specify where the content from text markup documents gets inserted and allows to add additional dynamic elements like navigation bars.

- **AST Rewriting**: Hooks into phase 2 described above. The advantage of using this hook over tweaking renderers is that changes made to the AST are always reflected in all rendered output formats. So if your custom logic is not specific to the output format, you should prefer this option. Writing a custom rewrite rule requires pattern matching on the AST node you are interested in, therefore, you may also want to read about **The Document AST**.

- **Overriding Renderers**: Hooks into phase 4 described above. It allows you to override the default renderer for one or more specific AST node types, while falling back to the built-in renderers for all others. Overriding renderers also requires pattern matching on the AST node you are interested in, therefore, you may also want to read about **The Document AST**.

Creating Templates

Laika comes with a lightweight template engine that allows to customize the output of the transformed markup documents.

When using a theme, creating a custom template is often not necessary, as the theme will have higher level configuration options that drive the look and feel of the output, based on its own internal templates.

When subtle tweaks to a theme are not sufficient or when working without a theme, adding custom templates will give you additional flexibility in controlling the output.

A template lets you specify where the content from text markup documents gets inserted and allows to add additional dynamic elements like navigation bars. It does not give control over how individual AST nodes from the markup document get rendered. For customizing this aspect, see **Overriding Renderers**.

Document Names and Locations

There are two kinds of template documents you can add to your input directories:

- Default templates that get applied to all markup documents that do not explicitly specify a template

- Additional templates that only get applied when a markup document explicitly refers to them

Default Templates

A default template for each output format can be placed into the root directory of the input tree. If you have multiple input directories, it needs to be placed into one of the roots.

The naming pattern for default templates is `default.template.<suffix>`. Concretely this would be:

- `default.template.html` for generating sites.

- `default.template.epub.xhtml` for EPUB.

- `default.template.fo` for PDF (since PDF uses XSL-FO as an interim format).

This template is then used for all markup documents unless overridden by one of the two mechanisms described in the following sections.

Overrides per Directory

A sub-directory may contain an overriding default template. It would be detected by the same naming pattern as the root default template, and would be applied to all markup documents in this directory and its sub-directories.

Overrides per Document

Finally, when you want to apply a custom template to individual markup documents, you can explicitly refer to it in the configuration header of the markup document:

```
{%
  laika.template = ../custom.template.html
%}
```

The path is interpreted relative to the document, but has to be somewhere within one of the specified input directories, as templates are parsed and cached upfront before getting applied to documents.

All non-default templates must have a name with the pattern `<name>.template.<suffix>` where `<name>` is anything other than `default` and `<suffix>` is the same matching suffix as for default templates.

Template Syntax

The following, minimal example shows a template for HTML output:

```html
<html>
  <head>
    <title>${cursor.currentDocument.title}</title>
  </head>
  <body>
    @:navigationTree { entries = [{ target = "/" }] }
    <div class="content">
      ${cursor.currentDocument.content}
    </div>
  </body>
</html>
```

Apart from raw content in the target format HTML, it contains:

- A substitution variable that inserts the title (`${cursor.currentDocument.title}`). See **Substitution Variables** for details.

- Another substitution variable that inserts the content of the markup document (`${cursor.currentDocument.content}`). See **Inserting Content from Markup Documents** for details.

- A directive `@:navigationTree` for rendering a navigation tree from the root (`"/"`) of the input tree. See **Directives** for details.

Inserting Content from Markup Documents

When a template gets applied to text markup documents, you need to explicitly specify where the content from those documents should be inserted.

Main Document Body

As shown in the example above, the substitution reference `${cursor.currentDocument.content}`
causes the main body of the associated text markup document to be rendered in that position.

The main body is the entire content of the markup document except for regions marked as fragments.

The internal mechanics of this insertion, which is technically just a merging of two ASTs, is described in
more detail in **Internal Representation**.

Document Fragments

In some cases you might want the rendered content from the markup document to appear in more than
just one place. You may have a central column in the template for the main content and a sidebar or footer
that needs to be populated separately.

The separate sections can be marked in the text markup with the `@:fragment` directive:

```
@:fragment(sidebar)

This content will be *parsed* like all other
content, but will be available separately from the document content.
The block elements extend until the `@:@` fence below.

Therefore this line still belongs to the fragment.

@:@

This line doesn't and will be part of the main document content.
```

Within templates you can then refer to this content with the substitution variable:

```
`${cursor.currentDocument.fragments.sidebar}`
```

The fragment names (`sidebar` in this case) can be freely chosen, they do not have any predefined meaning
within Laika.

Substitution Variables

The previous section already introduced the substitution variables for inserting content from markup
documents, but there are plenty of other predefined variables you can use, in addition to your own
definitions.

The syntax is part of Laika's HOCON support and matches the **HOCON spec**.

A mandatory reference can be inserted with the key of the variable enclosed between `${` and `}`:

```
${cursor.currentDocument.title}
```

If such a value is undefined, the transformation will fail (unless you tweaked the configuration for **Error Handling**).

An optional reference can be expressed by adding an additional ? right before the variable key:

```
${?cursor.currentDocument.title}
```

If an optional reference points to an undefined value, the reference in the template is substituted with an empty string.

For a full list of predefined values see **Substitution Variables** in the Reference section. For instructions on defining your own see **User-Defined Variables**.

Directives

Directives are Laika's mechanism for extending the syntax of templates or text markup without the need to implement a custom parser.

A directive always starts with an `@:`, the name of the directive and optionally attributes or body elements.

- Positional attributes can follow enclosed in (and), e.g. `@:fragment(sidebar)`.

- Named attributes in HOCON format can follow between { and } and can be any valid HOCON structure, e.g. `@:navigationTree { entries = [{ target = "/" }] }`.

- Body elements is regular markup or template content immediately after the directive declaration until a `@:@` fence, as shown in the example in **Document Fragments**. The AST for these parsed body elements will be passed to the directive implementation for processing.

- Positional and named attribute can theoretically both appear in the same directive, but is usually avoided for sakes of clarity and simplicity.

Some directive implementations require a set of mandatory attributes and may cause the transformation to fail should they be missing.

The most relevant directives for templates are those for generating navigation elements, like **Generating Navigation Trees** or **Breadcrumbs**.

For a full reference of the directives provided by Laika out of the box, see **Standard Directives**.

For instructions on how to write your own, see **Implementing Directives**.

Internal Representation

Templates get parsed into an AST just like text markup documents. This has several convenient consequences.

First, custom elements like directives can be implemented in a way that they produce AST nodes and not immediately string content in the output format. Therefore a single implementation of a template directive can usually be used for all supported output formats.

Secondly the implementation of "applying a template to a markup document" becomes close to trivial. It is merely the insertion of one AST at a particular node in another AST. Let's illustrate this with an example.

This is the template shown earlier in this chapter, further simplified to keep the AST output concise:

```
<html>
  <head>
    <title>${cursor.currentDocument.title}</title>
  </head>
  <body>
    ${cursor.currentDocument.content}
  </body>
</html>
```

The AST for this template looks like this:

```
TemplateRoot - TemplateSpans: 5
. TemplateString - '<html>|  <head>|    <title>'
. TemplateContextReference(cursor.currentDocument.title,true)
. TemplateString - '</title>|  </head>|  <body>|    '
. TemplateContextReference(cursor.currentDocument.content,true)
. TemplateString - '|  </body>|</html>'
```

The raw content gets represented by `TemplateString` nodes. The AST renderer truncates beyond a certain length and replaces newlines with | for putting more emphasis on the structure. The variable reference gets represented by a `TemplateContextReference` which will be resolved later.

More complex real-world scenarios will have additional node types, like those for representing a directive.

Next we create an equally minimal text markup document:

```
Headline
========

Some Text.
```

This produces just the Title node and a Paragraph in the resulting AST:

```
RootElement - Blocks: 2
. Title(Id(headline) + Styles(title)) - Spans: 1
. . Text - 'Headline'
. Paragraph - Spans: 1
. . Text - 'Some Text.'
```

This is just the tip of iceberg of available node types, for a general overview see **The Document AST**.

We then finally apply the template to the document. In a normal Laika transformation this happens automatically as the final step before rendering, but when working directly with the document model, you can alternatively call `applyTo` on a `TemplateDocument` instance.

The result will look like this:

```
RootElement - Blocks: 1
. TemplateRoot - TemplateSpans: 5
. . TemplateString - '<html>|  <head>|    <title>'
. . TemplateElement(0)
. . . SpanSequence - Spans: 1
. . . . Text - 'Headline'
. . TemplateString - '</title>|  </head>|  <body>|    '
. . EmbeddedRoot(4) - Blocks: 2
. . . Title(Id(headline) + Styles(title)) - Spans: 1
. . . . Text - 'Headline'
. . . Paragraph - Spans: 1
. . . . Text - 'Some Text.'
. . TemplateString - '|  </body>|</html>'
```

The two nodes which previously represented the two context references have been replaced by the corresponding AST from the parsed markup.

In case of the main content, the node `EmbeddedRoot` represents the insertion point of one AST into another.

This final AST is the model that will be passed on to renderers.

The Document AST

Laika decouples the semantics of the various markup formats and those of the supported output formats by representing every document in a generic AST between parsing and rendering. This allows to add custom processing logic only operating on the document structure itself, so it can be used with all supported input and output formats unchanged.

This chapter gives a brief overview over the hierarchy of the main base traits for AST nodes as well as short listings for about 80% of available node types.

Trait Hierarchy

All AST nodes extend one of the base traits from the hierarchy as well as optionally various additional mixins.

Providing a rich collection of traits that assist with classification and optional functionality helps with developing generic processing logic that does not need to know all available concrete types. You can, for example, collectively process all `BlockContainer` or `ListContainer` nodes without caring about their concrete implementation.

The traits are not sealed as the model is designed to be extensible.

Base Traits

At the top of the hierarchy the AST contains the following node types:
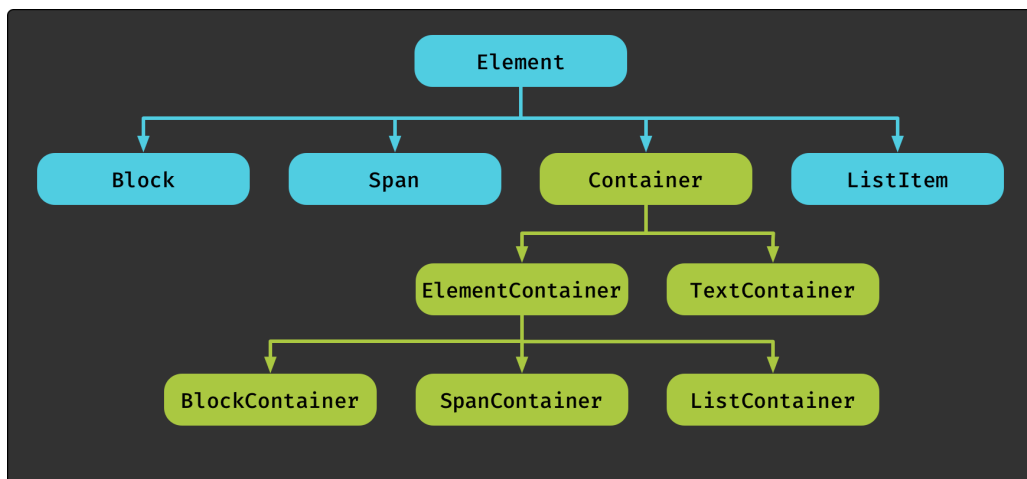
- `Element` is the base trait of the hierarchy. It extends `Product` and `Serializable` and is otherwise empty.

- `Customizable` is a mixin almost every node type supports that allows to optionally associate an id and/ or a set of style names to a node. The latter can be interpreted differently as hints by various renderers, in HTML they simply get rendered as class attributes for example. It also includes API to add or remove ids and style names polymorphically to any AST node.

- `Block` is one of the two major element types in the AST. Block level elements always start on a new line and in markup they often (but not always) require a preceding blank line. They can contain other block or span elements. It is similar to a `display: block` element in HTML.

- `Span` is the other of the two major element types in the AST. Span elements can start anywhere in the middle of a line or span multiple lines. They can contain other span elements, but no block elements. It is similar to a `display: inline` element in HTML.

- A `ListItem` can only occur as a child of a `ListContainer`. See **Lists** below for a list of available list types.

- While most other node types represent content parsed from text markup, a `TemplateSpan` represents a portion of a parsed template.

- The `Container` sub-hierarchy (explained in the next section) is not mutually exclusive to the base traits like `Block` or `Span`. Each container is always also one of the base types, but the mapping is not fixed and therefore not expressed in the hierarchy (e.g. a `SpanContainer` can be a `Span` or `Block` element itself).

Containers

Most element types in the AST are containers.

- `Container[+T]` is the base trait for all container types. It has a single property `content: T`.

- `TextContainer` is a simple `Container[String]`, a leaf node of the AST. See **Text Containers** below for a list of concrete types.

- `ElementContainer[+E <: Element]` is a `Container[Seq[E]]`. It is the base trait for all containers which hold other AST nodes.

  It mixes in `ElementTraversal` which has an API that allows to select child nodes (recursively) based on a predicate or partial function.

- `RewritableContainer` is the trait all containers need to mix in when they want to participate in an AST transformation. It has a single abstract method `rewriteChildren (rules: RewriteRules): Self`.

- `BlockContainer` is an `ElementContainer[Block]` and a `RewritableContainer`. See **Block Containers** below for a list of concrete types.

- `SpanContainer` is an `ElementContainer[Span]` and a `RewritableContainer`. See **Span Containers** below for a list of concrete types.

- `ListContainer` is an `ElementContainer[ListItem]` and a `RewritableContainer`. See **Lists** below for a list of concrete types.

Special Types

The first group of types are traits that can be mixed in to a concrete type:

- `Hidden` is a node type that holds metadata or link definitions and will be ignored by renderers.

- `Unresolved` is a node type that needs to be resolved during an AST transformation, for example a reference to a link definition by id. Having such a node type in the tree after all transformations are performed and right before rendering is an error and will cause the transformation to fail.

- `Invalid` is a node that has been inserted by a parser or an AST transformation rule to indicate a problem. Depending on the configuration the presence of such a node may either cause the transformation to fail, or get rendered as part of the page content for visual debugging. See **Error Handling** for details on how these nodes drive rendering and error reporting.

- `Fallback` is a node that holds an "alternative" node type representing the same content. When a parser or rewrite rule inserts a custom element type potentially not known by some renderers, it can offer a more common alternative, too.

- `SpanResolver` or `BlockResolver` nodes are special node types that know how to replace themselves during an AST transformation. Usually the node types and the rules that replace them during a transformation are decoupled. In cases where this is not necessary or not desired, this logic can be embedded in the node type itself. These traits have a `resolve` method that passes in a `DocumentCursor` that can be used to create the replacement node. It's useful for scenarios where producing the final node requires access to the configuration or other documents, which is not possible during the parsing phase. See **Cursors** below and the chapter on **AST Rewriting** for more details.

Finally there is a group of concrete types with special behaviour:

- `DocumentFragment` holds a named body element that is not supposed to be rendered with the main content. It may be used for populating template elements like sidebars and footers. See **Document Fragments** for details.

- `TargetFormat` holds content that should only be included for the specified target formats. All other renderers are supposed to ignore this node. As an example, you may want to render some content on the generated website, but not in the e-books (EPUB and PDF).

- `RuntimeMessage` is a node type that contains a message associated with a level. The presence of any message with a level of `Error` or higher usually causes the transformation to fail, unless you switched to visual debugging. See **Error Handling** for the relevant configuration options. Parser extensions, directives or rewrite rules that encounter errors can use this node type to report back on the issue in the exact location of the AST it occurred.

Container Elements

Since the document AST is a recursive structure most elements are either a container of other elements or a `TextContainer`. The few that are neither are listed in **Other Elements**.

The following lists describe the majority of the available types, only leaving out some of the more exotic options.

Lists

There are four different list types in the core AST model. All types listed below have a corresponding `ListItem` container, e.g. `BulletList` nodes hold `BulletListItem` children.

- `BulletList` represents a standard bullet list containing a definition of the bullet type alongside the list items.

- `EnumList` is an enumerated list containing a definition of the enumeration style alongside the list items. Supported enumeration styles are `Arabic`, `LowerRoman`, `UpperRoman`, `LowerAlpha` and `UpperAlpha`. reStructuredText supports all enumeration styles, Markdown only `Arabic`.

- `DefinitionList` is a list of definitions that associate terms with descriptions.

- `NavigationList` is a nested tree structure of internal or external links, often auto-generated from a particular node in the document tree.

Block Containers

- `RootElement` represents the root node of a document AST.

- `Section` represents a single section, containing a title and the block content of the section. This node type may contain further nested sections.

- `TitledBlock` associates a title with block content, but in contrast to a `Section` it will not contribute to auto-generated navigation structures for the document.

- `Figure` groups an image with a caption and an optional legend.

- `QuotedBlock` represent a quotation.

- `Footnote` represents a footnote that will render differently with each output format. In HTML footnotes will render in the location they were defined in, but can be styled differently. In PDF they become proper footnotes at the bottom of the page and in EPUB they become popups by default.

- `BlockSequence` is a generic sequence of blocks without any semantics attached.

Links and References

Most links and references are also a `SpanContainer` (for the link text), but a few are not. Span containers that are not links or references are listed in the next section.

The `Link` types differ from the `Reference` types in that the former are fully resolved, "ready to render", whereas the latter still need to be resolved based on the surrounding content or other documents.

`Reference` nodes only appear in the AST before the AST transformation step, where they will be either translated to a corresponding `Link` node or an `Invalid` node in case of errors.

For details on this functionality from a markup author's perspective, see **Navigation**.

Fully resolved `Link` nodes:

- `SpanLink` is a link associated to a sequence of spans, which may be text or images.

- `FootnoteLink` is a link to a footnote in the same document. The link text here is usually just a label or number, e.g. `[4]`.

`Reference` nodes that need to be resolved during AST rewriting:

- `PathReference` refers to a different document or section by a relative path, resolves to `SpanLink`.

- `LinkIdReference` refers either to the id of a URL defined elsewhere or directly to a section-id, resolves to `SpanLink`.

- `ImageIdReference` refers to the id of an image URL defined elsewhere, resolves to `Image`.

- `FootnoteReference` refers to a footnote in the same document, resolves to `FootnoteLink`. Supports special label functionality like auto-numbering.

- `ContextReference` encapsulates a reference in HOCON syntax, e.g. `${cursor.currentDocument.title}`. Resolves to whatever type the `Config` instance holds for the specified key.

Span Containers

- `Paragraph` is the basic span container type. Each block element without explicit markup that indicates a special type like a list or a header is represented by this node type.

- `Header` is a header element with an associated level.

- `Title` represents the first header of the document.

- `CodeBlock` represents a block of code and the name of the language. When Laika's built-in syntax highlighter is active and recognizes this language, the content will be a sequence of `CodeSpan` nodes with associated categories. Otherwise a single `CodeSpan` node will hold the entire content.

- `Emphasized`, `Strong`, `Deleted` and `Inserted` all contain a sequence of spans with the corresponding associated semantics as indicated by the markup.

- `SpanSequence` is a generic sequence of spans without any semantics attached.

Text Containers

- `Text` is the node type representing unformatted text.

- `Literal` is a literal span type.

- `LiteralBlock` is a literal block type that usually renders with whitespace preserved.

- `CodeSpan` is a span of text associated to a code category. Used in a `CodeBlock` where Laika's internal syntax highlighter has been applied.

- `Comment` is a comment in a markup document (not a comment in a code block). It is not a `Hidden` node as renderers for output formats that support comment syntax render this node.

Other Elements

This section lists the block and span elements that are not containers and the special elements representing templates.

Block Elements

Most block elements are either **Span Containers** or **Block Containers**, but a few are neither:

- The `Table` element represents a table structure with the help of its related types `Row`, `Cell`, `TableHead`, `TableBody`, `Caption` and `Columns`.

  Laika supports three table types out of the box: The grid table and simple table in reStructuredText and the table from GitHubFlavor for Markdown. All three types parse to the same model structure, but some of the simpler table types might only use a subset of the available node types.

- The `LinkDefinition` is a `Hidden` element that represents a link to an external or internal target mapped to an id that can be referenced by other elements.

  Both, the Markdown and reStructuredText parsers produce such an element, albeit with different syntax. In Markdown the syntax is `[id]: http://foo.com/` for example.

  All renderers are supposed to ignore this element and it normally gets removed during the AST transformation phase.

- The `PageBreak` element can be used to explicitly request a page break at the position of the node. Currently all renderers apart from the one for PDF ignore this element.

Span Elements

Most span elements are either **Span Containers** or **Text Containers**, but a few are neither:

- The `Image` element is a `Span` element that points to an image resource which may be internal or external. It supports optional size and title attributes.

  If you want to use an image as a block-level element you can wrap it in a `Paragraph`.

- The `LineBreak` represents an explicit line break in inline content.

Template Spans

Templates get parsed into an AST in a similar way as markup documents, but the model is much simpler. The most important element types are listed below.

- A `TemplateString` is a `TextContainer` that holds a raw template element in the target format (e.g. HTML).

- A `TemplateElement` is a wrapper around a `Block` or `Span` element from a markup AST.

  This allows the use of directives or other techniques that produce markup node types inside templates, so that the same directive can be used in template and markup documents.

- A `TemplateSpanSequence` is a generic sequence of `TemplateSpan` elements.

- The `TemplateRoot` node is the root element of every template, similar to how `RootElement` is the root of all markup documents.

- `EmbeddedRoot` is the node holding the merged content from the associated markup document. Usually one node of this type gets produced by every AST transformation.

AST Element Companions

Laika includes base traits called `BlockContainerCompanion` and `SpanContainerCompanion` for adding companion objects that provide convenient shortcuts for creating containers.

They allow to shorten constructor invocations for simple use cases, e.g. `Paragraph(Seq(Text("hello")))` can be changed to `Paragraph("hello")`. They also add an `empty` constructor and a vararg constructor for passing the content.

If you create a custom element type you can use these base traits to get all these shortcuts with a few lines:

```scala
object MyElement extends BlockContainerCompanion {
  type ContainerType = MyElement
  override protected def createBlockContainer (blocks: Seq[Block]) =
    MyElement(blocks)
}
```

Document Trees

So far we dealt with `Element` types which are used to represent the content of a single markup document or template.

When a transformation of inputs from an entire directory gets processed, the content gets assembled into a `DocumentTree`, consisting of nested `DocumentTree` nodes and leaf `Document` nodes.

The Document Type

This is the signature of the `Document` class:

```scala
case class Document (
  path: Path,
  content: RootElement,
  fragments: Map[String, Element] = Map.empty,
  config: Config = Config.empty,
  position: TreePosition = TreePosition.orphan
) extends DocumentStructure with TreeContent
```

- The `path` property holds the absolute, virtual path of the document inside the tree. It is virtual as no content in Laika has to originate from the file system. See **Virtual Tree Abstraction** for details.

- The `content` property holds the parsed content of the markup document. Its type `RootElement` is one of the types listed under **Block Containers** above.

- The `fragment` property holds a map of fragments, which are named body elements that are not supposed to be rendered with the main content. It may be used for populating template elements like sidebars and footers. See **Document Fragments** for details.

- The `config` property holds the configuration parsed from an optional HOCON header in the markup document. It is an empty instance if there is no such header. See **Laika's HOCON API** for details.

- The `position` property represents the position of the document in a tree. It can be used for functionality like auto-numbering. It is not populated if the transformation deals with a single document only.

The `DocumentStructure` mixin provides additional methods as shortcuts for selecting content from the document:

- `title:  Option[SpanSequence]` provides the title of the document, either from the configuration header or the first header in the markup. It is empty when none of the two are defined.

- `sections:  Seq[SectionInfo]` provides the hierarchical structure of sections, obtained from the headers inside the document and their assigned levels.

The DocumentTree Type

In a multi-input transformation all `Document` instances get assembled into a recursive structure of `DocumentTree` instances:

```scala
case class DocumentTree (
  path: Path,
  content: Seq[TreeContent],
  titleDocument: Option[Document] = None,
  templates: Seq[TemplateDocument] = Nil,
  config: Config = Config.empty,
  position: TreePosition = TreePosition.root
) extends TreeStructure with TreeContent
```

- Like with documents, the `path` property holds the absolute, virtual path of the document inside the tree. For the root tree it will be `Root`.

- The `content` property holds the child trees and documents of this tree. Its type `TreeContent` is a trait implemented by both `Document` and `DocumentTree`.

- The `titleDocument` property holds an optional title document for this tree. A tree often represents a logical structure like a chapter. It is used as a section headline in auto-generated site navigation as

well as the first content of this tree in linearized e-book output (EPUB or PDF). See **Title Documents** for details.

- The `templates` property holds the AST of all templates parsed inside this tree (excluding templates from child trees). The AST nodes it can hold are described in **Template Spans** above. See **Creating Templates** for more details on the template engine.

- The `config` property holds the configuration parsed from an optional HOCON document named `directory.conf`. It is an empty instance if there is no file. It can also be populated programmatically in cases where the content is generated and not loaded from the file system. See **Laika's HOCON API** for details.

- The `position` property represents the position of the document in a tree. It can be used for functionality like auto-numbering. It is not populated if the transformation deals with a single document only.

The `TreeStructure` mixin provides additional methods as shortcuts for selecting content from the tree:

- `selectSubtree`, `selectTemplate` and `selectDocument` all accept a `RelativePath` argument to select content from the tree structure, including content from sub-trees.

- `allDocuments: Seq[Document]` selects all documents contained in this tree, fetched recursively, depth-first. This is in contrast to the `content` property which selects documents and sub-trees on the current level.

- The `runtimeMessage` method selects any `RuntimeMessage` matching the specified filter. Useful for reporting errors and warnings in the parsed result.

Cursors

During an AST transformation the rewrite rule might require access to more than just the AST node passed to it. When it is a rule for resolving references or creating a table of contents, it needs access to the AST of other documents, too.

The `Cursor` type provides this access, but in contrast to the recursive `DocumentTree` which is a classic tree structure, it represents the tree from the perspective of the current document, with methods to navigate to parents, siblings or the root tree.

An instance of `DocumentCursor` is passed to rewrite rules for AST transformations (see **AST Rewriting**, and to directive implementations that request access to it (see **Implementing Directives**).

Let's look at some of its properties:

- `target: Document` represents the document this cursor points to.

- `parent: TreeCursor` represents the parent tree this document belongs to. A `TreeCursor` is similar to a `DocumentCursor`, just that its target is a `DocumentTree` instead.

- `root: RootCursor` represents the root of the entire tree.

- `previousDocument: Option[DocumentCursor]` and `nextDocument: Option[DocumentCursor]` represent its siblings. For a cursor pointing to the first document in a sub-tree (chapter), `previousDocument` will be empty, for the last document `nextDocument` will be empty.

- `flattenedSiblings.previousDocument` and `flattenedSiblings.nextDocument` are alternatives that look beyond the current sub-tree and flatten the hierarchy into a single list in depth-first traversal.

AST Transformation Phases

In most transformations the AST moves through three different phases between parsing and rendering:

1) The first shape will be the AST produced by the parsers for text markup or templates. Since parsers do not have access to the surrounding nodes or the configuration, some parsers for elements like links or navigation structures need to insert temporary node types.

2) After parsing of all participating documents and templates completes, the first AST transformation is performed. It resolves links, variables, footnotes, builds the document's section structure or generates a table of contents. These transformations are defined in rewrite rules. The library contains a basic set of rules internally for linking and navigation, but users can provide additional, custom rules. See **AST Rewriting** for details.

3) Finally the resolved AST representing the markup document is applied to the AST of the template. It is merely the insertion of one AST at a particular node in another AST.

The result obtained from step 3 is then passed to renderers. When rendering the same content to multiple output formats, the steps 1 and 2 are always only executed once. Only step 3 has to be repeated for each output format, as each format comes with its own templates.

AST Rewriting

The document tree in a Laika transformation is a generic representation of the document that does not contain any specific semantic or technical coupling to a concrete input or output format. This allows to add custom processing logic only operating on the document structure itself, so it can be used with all supported input and output formats unchanged.

Rewriting a tree means traversing it and replacing or removing some of its nodes. A tree model is immutable, so rewriting always creates a new tree, while reusing unmodified branches for efficiency.

Several of Laika's built-in features are implemented as rewrite rules internally, like the mechanism for resolving internal references or footnote auto-numbering. But it also designed to be a hook for processing on the application level.

How Rewrite Rules Work

A rewrite rule has the type `PartialFunction[T,  RewriteAction[T]]`. Laika offers a type alias `RewriteRule[T]` for convenience.

The rules are as follows:

- If the function is not defined for a particular element or it returns `Retain` the old element is kept in the tree.

- If the function returns `Replace(newElement)` this element is used in place of the old one.

- If the function returns `Remove` the old element is removed from the tree.

- Processing happens depth-first (bottom-up), so all nodes getting passed to this function already had their children getting processed.

- The tree is immutable, therefore new instances are returned when rewriting.

The following sections show the three ways to apply such a rule.

Applying a Rewrite Rule

The mechanism is slightly different, depending on whether you are using the sbt plugin or Laika embedded in an application. In the latter case you have two choices, one for hooking into a full transformation, the other for operating on nodes obtained by a separate parse operation. All three options are described below.

Using the sbt Plugin

The following example of an sbt build file shows how to turn each `Emphasized` node into a `Strong` node while processing everything else with default rules:

```
import laika.ast._

laikaExtensions += laikaSpanRewriteRule {
  case Emphasized(content, opts) => Replace(Strong(content, opts))
}
```

Using the Transformer API

When using the library API and all you want to do is to perform a full transformation from some input text to some output format, the Transformer API offers a hook to do this in one go, as a step in the transformation process.

Again we replace all `Emphasized` nodes with `Strong` nodes:

```
val transformer = Transformer
  .from(ReStructuredText)
  .to(HTML)
  .usingSpanRule {
    case Emphasized(content, opts) => Replace(Strong(content, opts))
  }.build
```

Working with the Tree Model

The final option is to use the `rewrite` method on individual documents or AST nodes.

The types `DocumentTree` and `Document` come with a `rewrite` method as well as most of the node types that can occur within a document (all nodes that mix in `RewritableContainer`).

Obtaining `Document` instances is usually achieved by splitting the parsing and rendering operations instead of using a full transformer. This is described in detail in **Separate Parsing and Rendering**.

Once again we are turning all `Emphasized` nodes in the text to `Strong` nodes for our example:

```
val doc: Document = ??? // obtained through the Parser API

val newDoc = doc.rewrite {
  case Emphasized(content, opts) => Replace(Strong(content, opts))
}
```

For a slightly more advanced example, let's assume you only want to replace `Emphasized` nodes inside headers. To accomplish this you need to nest a rewrite operation inside another one:

```
val newDoc = doc.rewrite {
  case h: Header => Some(h.rewriteChildren {
    case Emphasized(content, opts) => Replace(Strong(content, opts))
  })
}
```

Overriding Renderers

In some cases you might want to override the output of a renderer for a few types of AST nodes only, while keeping the default for the rest. Both the sbt plugin and the library API offer a hook to easily do that without modifying or extending the existing renderer.

In general this processing hook is intended for cases where the customization you intend to do is coupled to a specific output format. For any more generic processing logic, it is recommended to consider **AST Rewriting** instead, as that hook allows to replace or remove individual nodes in the AST before rendering. This way the same logic can be used for all supported output formats.

Function Signature

This is the signature of a custom renderer hook:

```
PartialFunction[(Formatter, Element), String]
```

`Formatter` is a generic type representing the formatting API which is different for each output format. For HTML it is `HTMLFormatterr`, for XSL-FO it is `FOFormatter`. It provides a helper APIs for rendering tags, adds the current indentation level after line breaks and knows how to render child elements.

`Element` is the base type of the document AST and represents the AST node to render, `String` is the render result in the expected target format.

See **The Document AST** for more details about the various node types

Defining a Render Function

This section explains how a render function is implemented and the subsequent sections show the three different ways to register such a function.

In the following example only the HTML output for emphasized text will be modified, adding a specific style class:

```
val renderer: PartialFunction[(HTMLFormatter, Element), String] = {
  case (fmt, Emphasized(content, opt)) =>
    fmt.element("em", opt, content, "class" -> "big")
}
```

For all node types where the partial function is not defined, the default renderer will be used.

Multiple custom renderers can be specified for the same transformation, they will be tried in the order you added them, falling back to the default in case none is defined for a specific node.

The `content` value above is of type `Seq[Span]`. A renderer should only ever render a single node and delegate to the formatter for rendering children. Only the formatter has a list of all installed render

extensions as well as the base renderer and will delegate to those functions where the partial function is defined for the child element.

Registering a Render Function

Since it is one of the most likely extension points used in user-code, there is a direct shortcut for passing it to the Laika configuration as shown below.

In case you want to combine it with other extensions, a render override can also be defined as part of an `ExtensionBundle` (see **The ExtensionBundle API** for details).

```
import laika.ast._

laikaSiteRenderers += laikaSiteRenderer {
  case (fmt, Emphasized(content, opt)) =>
    fmt.element("em", opt, content, "class" -> "big")
}
```

The Formatter APIs

Each formatter provides functionality that is specific to the target format. On top of that all formatters manage the indentation level after line breaks and know how to delegate to the relevant renderers for child elements.

TextFormatter

This is the base API supported by both the `XSL-FO` and `HTML` renderer, both of them adding several methods with rendering logic specific to that format.

- `newLine` renders a newline character followed by whitespace for the current level of indentation.

- `child` render a single child element.

- `childPerLine` renders a list of child elements, each on a separate line, with the current level of indentation.

- `indentedChildren` renders a list of child elements, each on a separate line, indented one level to the right from the current indentation level.

HTMLFormatter

This formatter supports all methods of the `TextFormatter` API shown above, and adds the following methods:

- `element` renders a tag where the specified list of child elements will be used to render the content of the tag

- `indentedElement` is similar to `element`, but renders the child elements one per line and indented to the right

- `textElement` renders an element with text content with special characters replaced by entities

- `emptyElement` renders an empty tag

Overview

Laika's Documentation comes with separate sections for "Customizing" and "Extending" Laika. While the line between the two is naturally quite blurry, the distinction is mostly meant to be between these two scenarios:

- Customizing Laika refers to the things you most like want to tweak and adjust in your own project.

- Extending Laika refers to the kind of re-usable extensions you might want to write as an in-house library or a 3rd-party open source extension.

This section deals with the latter scenario.

Transformation Phases

For a better understanding of the extension points listed below it is good to a have a rough idea about how a transformation in Laika is performed. It can be divided into 4 phases:

1) **The parsing step**. Text markup and template documents get parsed and translated into an internal AST. The AST is a generic abstraction of the document's structure and is not tied to any specific semantics of a particular input or output format.

2) **The AST transformation**. The original AST is only what each parser for the corresponding block or inline element can process locally, without access to other nodes or even other documents. One of the advantages of this design, apart from separation of concerns, is that parsers can run in parallel. As a consequence nodes like internal references or auto-numbered footnotes require further processing with access to a `DocumentCursor` that allows to access content from anywhere in the input tree.

3) **Applying templates to markup documents**. Since both are just AST structures, this step is merely a second AST transformation. The AST representing the markup document will be inserted into the node of the template AST that holds a reference to the content.

4) **Rendering**. As the last step the final AST obtained from the two previous transformation steps will get rendered to one or more output format. This is the only step specific to a particular output format, meaning the same AST structure obtained in 3) will get used as the input for the renderers of all formats.

The ExtensionBundle API

Hooks into the 4 transformation phases described above as well as other customization hooks can be used by implementing an `ExtensionBundle`. See **ExtensionBundle** for its API documentation.

The trait comes with empty default implementations for most of its properties, so that you only need to override the ones you intend to use.

```
object MyExtensions extends ExtensionBundle {

  override val docTypeMatcher: PartialFunction[Path, DocumentType] = ???

  // ... optionally other customizations
}
```

Such a bundle can then be passed to the transformer:

```
laikaExtensions := Seq(
  GitHubFlavor,
  MyExtensions
)
```

You've probably already seen examples for specifying `GitHubFlavor` or `SyntaxHighlighting` extensions in this way. These are implementations of `ExtensionBundle`, too, and come bundled with the `laika-core` module.

The Theme API

Creating themes expands on the extension hooks provided by bundles and adds the capability of pre-populating the input tree with templates and styles. It allows to offer users ready-to-use rendering styles without the need to craft their own templates, CSS or JavaScript.

Laika also comes with a lightweight default theme called Helium, and in many cases just tweaking its settings might give you enough flexibility for achieving the desired look & feel. The theme comes with default styles for the three main output formats of Laika: web site, EPUB and PDF. Configuring Helium is covered in detail in the chapter **Theme Settings**.

If you need full control over all aspect of the design, see the chapter **Creating Themes** for instructions. The remainder of this chapter deals with the functionality of the `ExtensionBundle` API introduced above.

Extending Markup Syntax

Laika offers two major options for extending the native syntax of a text markup language:

- **Implementing Directives** is an option that allows to quickly add custom functionality without implementing a custom parser. Laika comes bundled with a handful of directives which are documented in **Standard Directives**.

- **Writing Parser Extensions** is the alternative for cases where you have special requirements for the syntax so that the convenient path of writing a directive is not feasible. In this case you also need to become familiar with **Laika's Parser Combinators**.

Technically directives hook into phase 2 of a transformation while parser extensions live in phase 1. Parsing of directives is performed by Laika's built-in parsers as they all have a common syntax. A directive

implementation is then invoked during AST transformation, receiving all the attributes and body elements the user specified to produce a new AST node to insert.

Additional Syntax Highlighters

Laika has its own built-in syntax highlighters (based on its parser combinators). See **Syntax Highlighting** for a list of languages supported out of the box.

If you want to use a language that is not supported yet, you can either decide to use an external tool like `highlight.js` for this task (Laika renders code elements with attributes compatible with that tool) or write your own implementation for Laika's highlighters.

If you look at the existing implementations, you'll notice that some of them are almost purely declarative, as the library provides a set of building blocks for parsing the most common literal formats or comments.

But in most cases you'd also need to get familiar with **Laika's Parser Combinators**. See **Adding Syntax Highlighters** for more details.

Additional Markup or Output Formats

Apart from extending or customizing existing markup or output formats, you can also add support for additional formats. If you want to support a different text markup language like ASCIIDoc or Textile, or if you want to produce Word output or slide formats, you can implement a `MarkupFormat` or `RenderFormat` as shown in **New Markup or Output Formats**.

Customization Hooks

Two of the most commonly used hooks in the `ExtensionBundle` API are described in the "Customizing Laika" section of the manual:

- **AST Rewriting**, a hook into phase 2 that allows to replace or remove nodes from the AST model between parsing and rendering.

- **Overriding Renderers**, a hook into phase 4 that allows to override the rendered output for specific AST nodes.

There are two further hooks that drive more low-level functionality:

- The `docTypeMatcher` property in `ExtensionBundle` controls how a virtual path is used to determine the document type (e.g. markup document vs. template vs. configuration file).

- The `slugBuilder` property in `ExtensionBundle` controls how the text from a section headline is translated to a slug for element ids. It is a simple function `String => String`.

Replacing Internal Parsers

Laika does not only come with parsers for supported text markup languages. It uses additional parsers for its HOCON configuration support, for parsing templates and for its "CSS for PDF" functionality.

Although probably the least likely extension point you need to use, these three parsers can all be replaced by implementing the respective properties in the ParserBundle API: `configProvider`, `templateParser` and `styleSheetParser`. Such a bundle can then be declared in an `ExtensionBundle` with the `parsers` property.

Creating Themes

Creating your own theme is a way to build an alternative for Laika's default theme called Helium in cases where its feature set or look and feel is not meeting your requirements.

The effort of designing and implementing a theme is justified if you plan to re-use it across multiple projects, either as an in-house library or published as Open Source for the entire community.

If you just want to create a custom design for a single project, it's easier to just install an empty theme and simply place all necessary templates and CSS files in your regular input directory.

```
laikaTheme := Theme.empty
```

The Theme Trait

Before describing the typical steps for implementing a theme, let's have a quick look at the (simple) `Theme` trait, to get an idea of what a theme represents in technical terms.

```scala
trait Theme[F[_]] {

  def inputs: InputTree[F]

  def extensions: Seq[ExtensionBundle]

  def treeProcessor: Format => TreeProcessor[F]

}

object Theme {

  type TreeProcessor[F[_]] = Kleisli[F, ParsedTree[F], ParsedTree[F]]

}
```

The `inputs` property is the most important aspect of the trait, as it allows to pre-populate the input tree with templates and styles. See **Generating or Loading Documents** below for details.

The `extensions` property enables theme authors to "auto-install" one or more extension bundles that are required for the theme's feature set. For simple themes this collection may be empty. You can use any existing 3rd-party bundle or create your own.

For example the default Helium theme uses this option to install a few custom renderers that deal with UI aspects not covered by Laika Core, like decorating callouts with one of Helium's icons.

Bundles are not covered in this chapter as this extension point is documented in **The ExtensionBundle API**.

A bundle that is automatically installed as part of the theme should be essential for the theme to function. If it is opt-in, your library should make it available separately so that the user can mix and match.

111

Finally, the `treeProcessor` property is another optional hook that allows to inspect and modify the document AST between parsing and rendering. This can be useful if some of your theme's configuration logic needs to look at the input documents or configuration supplied by the user before generating the final AST. It is a function that expects the output format, so you can create separate logic for producing the AST for the site, EPUB and PDF output.

Designing a Configuration API

Every theme should have a configuration API that allows to tweak the look and feel or add metadata or custom links. It's usually the only public API of your library to reduce the likeliness of issues with binary compatibility.

You can have a look at Helium's API documented in **Theme Settings** for inspiration. Such an API usually covers some or all of the following aspects:

- Configure font resources for embedding

- Tweak colors, font sizes, line height and layout of the pages

- Define metadata like language, authors, description

- Configure special pages like table of contents, download or landing pages

- Add custom links to navigation bars

- Specify cover images for EPUB and PDF documents

The package `laika.theme` contains a few base types that you can reuse for defining some common types:

- The `Color` type gives you the `hex` and `rgb` methods for defining colors.

- `LengthUnit` offers constructors for all length units permitted in CSS, e.g. `px(12)` or `em(1.5)`.

- The `FontDefinition` type allows to define fonts for embedding. Adding these instances to the final base configuration will trigger the embedding in EPUB and PDF documents without any further implementation within your theme.

Another aspect users might appreciate is if you allow to define most options separately per output format. Users might pick a more colorful design for the site for example, but switch to a more black-and-white feel for the PDF output so that the content looks good when printed.

The Helium API solves this by requiring a selector in front of all configuration methods which is either `all`, `site`, `epub` or `pdf`:

```
val theme = Helium.defaults
  .all.metadata(
    title = Some("Project Name"),
    language = Some("de"),
  )
  .epub.navigationDepth(4)
  .pdf.navigationDepth(4)
  .build
```

Generating or Loading Documents

The one aspect probably all theme implementations are dealing with is to pre-populate the input tree with default templates and styles. For this you can use the same `InputTree` API that users can use if they need full flexibility for defining their inputs.

> ⚠️
>
> While Laika offers the full API as for user inputs for maximum flexibility and consistency, it is recommended to avoid any file system resources by using `addFile` or `addDirectory`.
>
> This might be acceptable for a library shared in-house where you can rely on a specific project setup, but for a public library it is most convenient for users when the theme is just an additional dependency and does not require any additional setup.

Templates

Each theme supporting the three major output formats (site, EPUB, PDF) would come with at least three template documents: one default template for each format. Additionally a theme can provide opt-in templates that users can explicitly select in the configuration header of a markup document with (`laika.template = /<theme-dir>/<name>-template.html`), where `theme-dir` is the directory your theme files are generated into.

There are two approaches you can choose from:

- Generate the entire template as an in-memory string based on the user's theme configuration and add the result to the `InputTreeBuilder`:

  ```
  val builder: InputTreeBuilder[F] = ???
  val templateString: String = MyTemplateGenerator.generate(config)
  builder.addString(templateString, DefaultTemplatePath.forHTML)
  ```

  This might be quite convenient to implement, but has the disadvantage that the user does not have a way to look at the default template in case she wants to place a modified copy into the input tree.

- Or place the entire default template into the resource folder of your library and load it from there:

```
val builder: InputTreeBuilder[F] = ???
val resourcePath = "my-theme/templates/default.template.html"
builder.addClasspathResource(resourcePath, DefaultTemplatePath.forHTML)
```

In this case you would use Laika's template syntax to access your theme's configuration, usually mostly by using the following features:

- HOCON substitution references in the form of `${myTheme.site.fontFamilies}`. For details see **Generating Configuration for Templates** below.

- Laika directives, `@:for` to loop over content placed into the configuration and `@:if` for conditionally including content.

In this approach your documentation can point to the source of the template in your repository, making it easier for the user to use a modified copy if required.

For details on the supported template syntax, see **Creating Templates**.

Generating Configuration for Templates

When using the second approach described in the section above where you use directives and substitution references inside your templates, the base configuration for the transformer needs to be pre-populated with all the corresponding values.

This can be achieved using Laika's `ConfigBuilder` API which allows to programmatically construct `Config` instances, which are normally obtained by parsing HOCON. The builder accepts all the types supported in JSON/HOCON, e.g. Strings, Numbers, Arrays and Objects, but also, as a Laika extension, the inclusion of AST nodes.

Using AST nodes has the advantage that you do not have to pre-render the output for all formats. If it is a node type supported by Laika Core it is already known to all renderers, allowing you to reduce the boilerplate and stringly logic of rendering the format directly.

The below example shows how the `ThemeBuilder` API can be used to pre-populate the transformer configuration:

```
val logo = Logo.internal(
  path = Root / "logo.png",
  alt = Some("Project Logo")
)

val baseConfig = ConfigBuilder.empty
  .withValue("theme-name.logo", logo)
  .build

ThemeBuilder("Theme Name")
  .addBaseConfig(baseConfig)
  .build
```

It defines a logo AST element, based on the virtual path `Root / "logo.png"` and associates it with the key `theme-name.logo`. Finally it passes the configuration to the theme builder, making the logo available for templates via a substitution reference (`${theme-name.logo}`).

The indirection via the configuration key means that even if the user customizes the default templates of the theme you created, these references can still be used by the end user.

Of course the above example is a minimal excerpt of a typical theme builder, which would normally add more keys to the configuration and also use the theme builder to pre-populate templates and styles.

CSS

The second content type you would most likely include is CSS generated based on user configuration. Here the most convenient approach might be to place static CSS files into the resource folder of your library, and use CSS variables to capture all aspects which the user can configure. This is the approach that Helium has also chosen.

```
val builder: InputTreeBuilder[F] = ???
val resourcePath = "my-theme/css/theme.css"
val vars: String = MyCSSVarsGenerator.generate(config)
builder
  .addString(vars, Root / "my-theme" / "vars.css")
  .addClasspathResource(resourcePath, Root / "my-theme" / "theme.css")
```

Fonts

If a theme supports EPUB or PDF output, it would be convenient for the user if a theme includes a set of default fonts. Otherwise PDF output would always be based on the few standard fonts that are available for every PDF generator, which might be somewhat limiting stylistically.

Ensure that the fonts your are using have a license that allows for redistribution as part of a library. Beware that some web fonts might allow linking the font in websites for free, but not embedding them into EPUB or PDF documents. Ideally the license should also not require the users of the theme to add an attribution to each page.

When including font defaults for convenience, the theme's configuration API should always allow for their replacement. The API should accept a sequence of `FontDefinition` instances that define the fonts to be embedded.

These definitions can then be passed to the base configuration of the theme (which will be merged with the user configuration):

```
val fonts: Seq[FontDefinition] = ???
val baseConfig = ConfigBuilder.empty
  .withValue("laika.epub.fonts", fonts)
  .withValue("laika.pdf.fonts", fonts)
  .build

ThemeBuilder("Theme Name")
  .addBaseConfig(baseConfig)
  .build
```

Of course, like with Laika's default Helium theme, you can allow to define different fonts for EPUB and PDF.

Constructing a Theme Instance

Finally all the templates, styles, configuration and fonts that you gather from the user's theme configuration or from your defaults if omitted, need to be assembled into a theme provider instance.

This step will be very different for each theme depending on its feature set, so we just show Laika's own Helium theme builder as an example:

```
class HeliumThemeBuilder (helium: Helium) extends ThemeProvider {

  def build[F[_]: Sync: Runtime]: Resource[F, Theme[F]] = {

    import helium._

    val treeProcessor = new HeliumTreeProcessor[F](helium)
    val htmlOverrides = HeliumRenderOverrides
                          .forHTML(siteSettings.layout.anchorPlacement)

    ThemeBuilder("Helium")
      .addInputs(HeliumInputBuilder.build(helium))
      .addBaseConfig(ConfigGenerator.populateConfig(helium))
      .addRewriteRules(HeliumRewriteRules.build(helium))
      .addRenderOverrides(HTML.Overrides(htmlOverrides))
      .addRenderOverrides(XSLFO.Overrides(HeliumRenderOverrides.forPDF))
      .processTree(treeProcessor.forHTML, HTML)
      .processTree(treeProcessor.forAllFormats)
      .build

  }
}
```

As you can see, it is a very tiny class as it delegates to various other builders to properly separate concerns. Let's just highlight the most important aspects:

- Your theme builder must implement `ThemeProvider` as that is ultimately the type that can be passed to the library's transformer API or to the `laikaTheme` sbt setting. An instance of this type is what the final call to `build` in your configuration API should produce.

- The `build` method will be invoked by Laika when the transformer resource is created. It passes in the effect type as it is in the hand of the end user to select the concrete implementation, any implementation of `cats.Sync` is supported. It also provides a `Runtime` instance, which is a Laika type that encapsulates user runtime configuration (the execution contexts to use, level of parallelism, etc.). It has an API that allows to call `runBlocking` or `runParallel` which will use the user provided contexts.

- The `ThemeBuilder` is an optional API, but usually recommended as it reduces boilerplate. The only requirement is that the `build` method returns a cats-effect `Resource[F, Theme[F]]`. In the case of Helium, you can see that the API is used to:

  a) Pre-populate the input tree with documents (templates, styles, JavaScript) via `addInputs`, as described in **Generating or Loading Documents** above.

  b) Pre-populate the transformer configuration (e.g. with text or AST nodes that can be referenced in templates) via `addBaseConfig` as shown in **Generating Configuration for Templates** above.

  c) Customize the rendering of some AST nodes with theme-specific features via `addRenderOverrides`. In Helium this is used for rendering tabs for user-selectable content, anchors for copying section links, and for adding custom icons to callouts. See **Overriding Renderers** for details.

  d) Add AST rewrite rules via `addRewriteRules`, which is somewhat more low-level and allows to remove or swap AST nodes between parsing and rendering. See **AST Rewriting** for details on this topic.

  e) Finally, it adds or removes additional documents via `processTree` which allows for a final modification of the document tree per output format, e.g. to remove a JavaScript file that your theme from the EPUB output which is only intended for websites.

Publishing a Theme

Since a theme is just a dependency, you can publish it like any other library.

As a minimum set of documentation it's recommended to include:

- Listing all licenses, both the one you are using for the theme implementation as well as potentially 3rd-party licenses for any font or icon resources you might include.

- Document the supported output formats. Ideally themes support Laika's main target formats, HTML, EPUB and PDF, but if you want to focus on fewer or different formats, let your users know.

- Document the compatible Laika versions. Laika is still in the 0.x version range, so some breaking changes are still expected before reaching the 1.0 milestone. But the big rewrite phases are a thing of

the past now, so you won't be exposed to massive disruptions. A 1.0 version is expected at some point after the 0.18 or 0.19 releases (we are at 0.16 right now).

- Document your theme configuration API.

Finally, let us know about your theme! We are happy to add links and short descriptions to Laika's documentation, so that users know which 3rd-party alternatives exist.

Implementing Directives

Implementing a directive is one of two possible ways to extend the syntax of a text markup language.

The second option is **Writing Parser Extensions**, but in many cases the directive approach offers more convenience. It is based on a common syntax for declaring a directive and its attributes and body elements. Therefore directives can be implemented without writing a custom parser and without getting familiar with Laika's parser combinators.

There are usually only two scenarios where you may prefer to write a parser extension instead:

- You have very special requirements for the syntax. If, for example, you want to design a new table markup format, this is quite impossible to express with the fixed syntax of a directive.

- You want the most concise syntax for elements that are used frequently. One example is to support short references to issues in a ticket system in the format `#123`. The shortest syntax that is possible with directives would be something like `@:ticket(123)` which you may still find too verbose.

Anatomy of a Directive

The following diagram shows a markup directive with all its optional elements present:
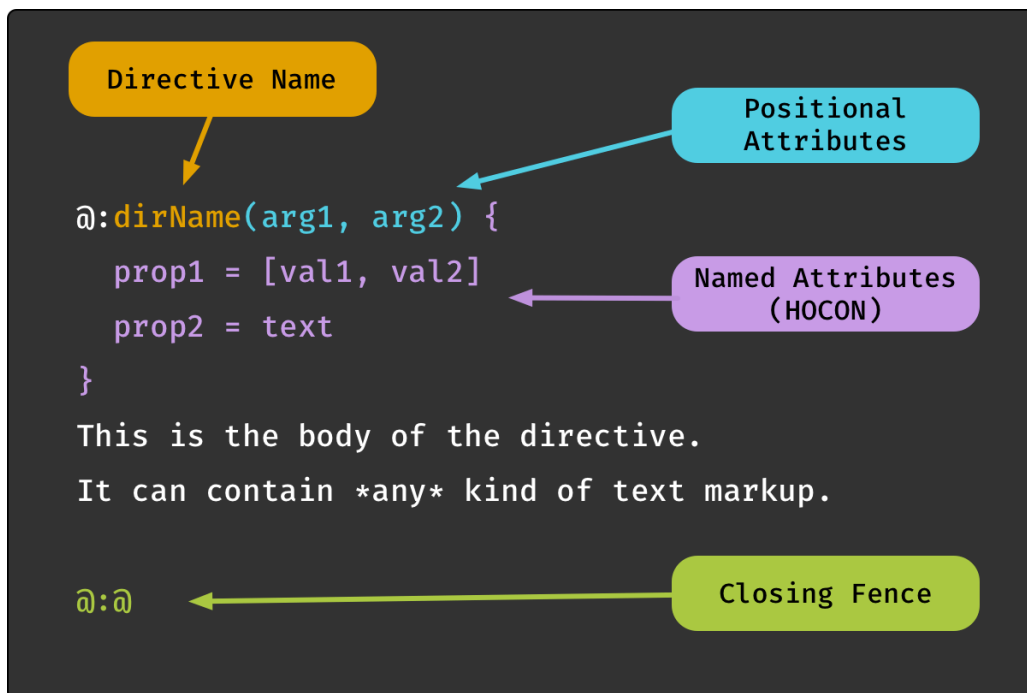
It is quite abstract and contrived, because any real-world directive would not use all optional elements at once.

Let's discuss the directive parts one by one:

- The **directive name** part consists of the character sequence `@:` followed by an alphanumeric name. It is the only required part, and in fact some of the built-in directives of the library like `@:pageBreak` do not accept any attributes or body elements. The convention is that directive names are camel case. Their main task is to link the markup to the corresponding directive implementation.

- **Positional Attributes** can follow the name declaration. They are enclosed in parentheses and contain a comma-separated list of values, mimicking the syntax of function invocations.

  They are best suited for directives that either accept a single attribute or a list of attributes of the *same type*. An example would be a list of style names to apply to the body. For multiple attributes with different types and meanings, it is recommended to use named attributes instead.

- **Named Attributes (HOCON)** can follow the name declaration and the optional positional attributes. They are enclosed in braces { and } as HOCON is a superset of JSON.

  They offer the full flexibility of the HOCON standard, like nested keys, array elements and substitution references (`${...}`) which allow markup authors to refer to any variable in page/directory/global configuration when defining an attribute.

- The **Directive Body** can follow the name declaration and any optional attribute blocks. It is a region of regular markup content demarcated by a closing fence `@:@`. With block directives the closing fence has to be placed on an otherwise empty line, with all other directive kinds it can appear anywhere.

  The author of a directive implementation can decide whether the body part should be processed as a raw string, or as pre-parsed AST nodes. In the latter case the type of the body passed to the implementation will be `Seq[Span]`, `Seq[Block]` or `Seq[TemplateSpan]`, depending on the type of the directive.

Finally, let's also examine the anatomy of a directive implementation:
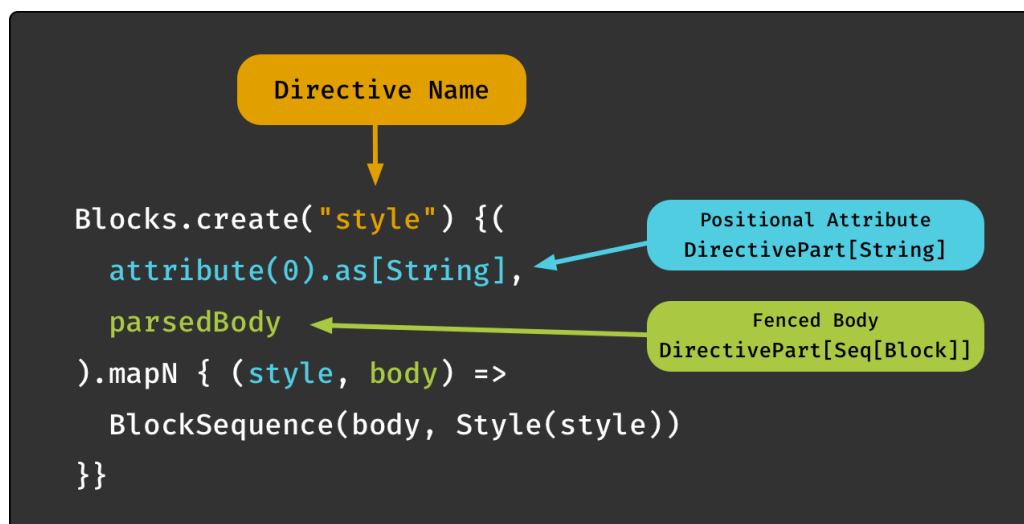
We'll go through this line by line:

- The imports you need depend on the directive type, you need to choose the matching DSL which is either `Blocks.dsl._`, `Spans.dsl._` or `Templates.dsl._`. You also need `cats.implicits._` for the convenient `mapN` method.

- The entry point for creating a directive again depends on the type, it's either `Blocks.create`, `Spans.create` or `Templates.create`. The string passed to this method is the name of the directive without the leading `@:`.

- Next you need to declare the directive parts that you expect, whether they are required or optional, and which type they should be converted to. There are combinators for **Directive Attributes**, **Directive Body**, **Access to the Parser** and **Access to the Document Cursor**.

- Finally you pass the function the engine should invoke after validating and converting all expected directive parts.

  The function will not be invoked at all in case of missing required parts or type conversion errors, instead an invalid node will inserted into the AST which will be handled depending on your **Error Handling** configuration. This makes it quite convenient for directive authors as it removes a lot of boilerplate.

  When the directive is valid, the parameters of the function will be of the type matching your spec, in our example `String` and `Seq[Block]`. The value the directive function produces must be an AST node matching the type of the directive, `Span`, `Block` or `TemplateSpan`.

Further below we'll show a **Sample Directive** based on a real world scenario.

Directive Types

Due to differences in the node types they produce and the location they are allowed to appear in, there is a distinction between seven directive types:

- **Template Directives** that can be used anywhere in templates, which produce `TemplateSpan` elements.

- **Block Directives** for block elements in markup, which produce `Block` elements.

- **Span Directives** for inline elements in markup, which produce `Span` elements.

- 3 types of **Separator Directives** which can be used to separate the body elements of any of the other three directive types and can produce any kind of result type.

- **Link Directives**, a special type of span directives providing shortcuts for directives that produce link nodes.

If you want to create a directive that can be used in all three locations you can usually easily reuse parts of your logic. If your directive version for inline elements generates a simple `SpanLink` node for example, you can wrap that same node in a `Paragraph` block node for a block-level directive.

The three main directive types for blocks, spans and templates are similar enough that we avoid the repetition of documenting them separately. Instead the following sections **Sample Directive**, **Directive**

Attributes, **Directive Body**, **Access to the Parser** and **Access to the Document Cursor** apply to all of these 3 main types.

Separator Directives and **Link Directives** on the other hand are quite different, which is why they come with their own documentation section.

Directives Provided by Laika

Even though this chapter is about implementing your own directives, it's worthwhile to briefly mention that Laika comes with a handful of directives out of the box.

First you might want to avoid to reinvent the wheel and ensure the directive idea you had is not already covered. Secondly you could look at the existing directive implementations as examples and guidelines for creating your own.

All available directives are documented in the chapter **Standard Directives**, their source code can be found **here**

Sample Directive

Like in the **Writing Parser Extensions** chapter our example implementation is going to provide a shortcut for a reference to a ticket system via its number. While the custom parser did this with the concise syntax #123, a directive has to stick to the standard directive syntax, for which we'll pick @:ticket(123).

Directive Implementation

Let's walk through the implementation of our little ticket directive:

```
import Spans.dsl._

val ticketDirective = Spans.create("ticket") {
  attribute(0).as[Int].map { ticketNo =>
    val url = s"http://our-tracker.com/$ticketNo"
    SpanLink(Seq(Text("#" + ticketNo)), ExternalTarget(url))
  }
}
```

- With `Spans.create("ticket")` we specify the name of the directive without the leading @: prefix.

- `attribute(0)` specifies a single, positional attribute (which has to appear between parenthesis). See **Directive Attributes** for an overview of all the other options.

- `as[Int]` converts the attribute to an Int before passing it to the directive. When the attribute is missing or not a valid integer, the directive fails and the function passed to `map` will never be invoked There is a range of decoders that you can use out of the box, but you can also provide your own `ConfigDecoder`.

- We are not calling `.optional` on the specified attribute, which means it is mandatory by default and the directive will fail if the user omits it (e.g. by just typing @:ticket).

- We finally create a `SpanLink` node (which implements `Span`) as the result. We use `# + num` as the link text and then build the URL as an external target.

This first example hard-codes the base URL. If your directive is for internal use, this may not be a problem. But if you require a configurable base URL, we later show an enhanced example that has access to the configuration.

Directive Registration

For bundling all your Laika directives, you need to extend `DirectiveRegistry`. This is a sub-trait of `ExtensionBundle`.

For our case we only need to register our ticket directive, which is a span directive:

```scala
object MyDirectives extends DirectiveRegistry {
  val spanDirectives = Seq(ticketDirective)
  val blockDirectives = Seq()
  val templateDirectives = Seq()
  val linkDirectives = Seq()
}
```

Finally we need to register our registry together with any built-in extensions you may use:

```scala
laikaExtensions := Seq(
  GitHubFlavor,
  MyDirectives
)
```

We can now use our `ticket` directive in Markdown or reStructuredText files!

Access to Configuration

This will enhance the previous example by making the base URL configurable.

One of the combinators we can use when defining the directive can ask for a document cursor to be provided alongside the expected attribute value:

```scala
import Spans.dsl._

val ticketDirective = Spans.create("ticket") {
  (attribute(0).as[Int], cursor).mapN { (num, cursor) =>
    cursor.config.get[String]("ticket.baseURL").fold(
      error => InvalidElement(s"Invalid base URL: $error", "#" + num).asSpan,
      baseURL => SpanLink(Seq(Text("#"+num)), ExternalTarget(s"$baseURL$num"))
    )
  }
}
```

The `DocumentCursor` instance passed to our function provides access to the project configuration which we use in this example, but also to all ASTs of the input tree, including other documents. It can therefore be used for advanced functionality like producing a table of contents.

The API of the `cursor.config` property is documented in **Config**.

In our case we expect a string, but we also need to handle errors now, as the access might fail when the value is missing or it's not a string. We return an `InvalidElement` for errors, which is a useful kind of AST node as it allows the user to control the error handling. The presence of such an element will by default cause the transformation to fail with the provided error message shown alongside any other errors encountered. But users can also switch to a "visual debugging" mode by rendering all errors in place. See **Error Handling** for details.

With this change in place, the user can now provide the base URL in the builder of the `Transformer`:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue("ticket.baseURL", "https://example.com/issues")
```

The registration steps are identical to the previous example.

Directive Attributes

You can use two types of attributes: positional attributes between parenthesis, and named attributes in the HOCON format between curly braces.

See **Anatomy of a Directive** for an overview and a full syntax example.

Positional Attributes

Positional attributes can follow the name declaration. They are enclosed in parentheses and contain a comma-separated list of values, mimicking the syntax of function invocations.

They are best suited for directives that either accept a single attribute or a list of attributes of the *same type*. An example would be a list of style names to apply to the body. For multiple attributes with different types and meanings, it is recommended to use named attributes instead.

Markup example:

```
@:format(html, epub)
```

Combinators:

- `attribute(Int)` for accessing a single attribute, e.g. `attribute(0)` for the first. Like with named attributes, you can turn them into **Optional Attributes** or apply **Type Conversions**.

- `positionalAttributes` for accessing all positional attributes as a list of values. Applying a type conversion like `.as[Int]` in this case changes the result to be `Seq[Int]`.

Combinator Example:

```scala
val ticketDirective = Spans.create("directive-name") {
  attribute(0).as[Int].map { ticketNo =>
    ??? // produce AST span node
  }
}
```

Named Attributes (HOCON)

Named Attributes in HOCON format can follow the name declaration and the optional positional attributes. They are enclosed in braces { and } as HOCON is a superset of JSON.

They offer the full flexibility of the HOCON standard, like nested keys, array elements and substitution references (`${...}`) which allow markup authors to refer to any variable in page/directory/global configuration when defining an attribute.

Markup example:

```
@:image { intrinsicWidth = 280px, intrinsicHeight = 220px }
```

Combinators:

- `attribute(String)` for accessing a single attribute by name, e.g. `attribute("width")` for the `width` attribute in our example. Like with positional attributes, you can turn them into **Optional Attributes** or apply **Type Conversions**.

- `allAttributes` for accessing all attributes as a HOCON `Config` instance, Laika's configuration API. This way you can manually inspect the values it contains. This naturally leaves the burden of validation and error handling (e.g. for required values or type conversions) with the implementor of the directive. It should therefore only be used when this level of flexibility is really required. See **Config** for the API documentation.

Combinator Example:

We'll use the `allAttributes` combinator together with the one for accessing body elements:

```scala
val directive = Spans.create("custom") {
  (allAttributes, parsedBody).mapN { (attributes, bodyContent) =>
    val path = attributes.getOpt[Path]("filePath")
    val index = attributes.getOpt[Int]("index")
    ??? // produce AST span node
  }
}
```

Optional Attributes

Positional and named attributes can be marked as optional.

Combinator Example:

```
attribute("width").as[Int].optional
```

The parameter type of your directive function changes accordingly, from `T` to `Option[T]` where `T` is either the type returned by your converter (see below) or the default type.

Type Conversions

You can specify a decoder for all attributes with the `as[T]` method:

```
attribute("depth").as[Int].optional
```

Without a decoder the result type would be `ConfigValue` which is a data structure similar to those of popular JSON libraries. But that type is rarely the most convenient type, which is why Laika defines some basic decoders out of the box, for `String`, `Int`, `Double`, `Boolean`, `Date` and `Path` as well as `Seq` and `Map` of containing these types.

The decoders will cause a directive to fail in case a type conversion does not succeed.

You can define your own by implementing `ConfigDecoder[T]`.

Inheritance

By default directives only accept attributes defined right in the attribute section of the directive to avoid name clashes with attributes in other scopes. If you want to explicitly enable inheritance, so that a user can define default values for attributes either programmatically in the global configuration or in headers of the markup or template files, you can set the `inherited` flag:

Combinator Example:

```
attribute("width").as[Int].inherited
```

With this flag set, the `width` attribute can be inherited from other scopes if it is not defined in the directive itself.

Directive Body

A directive body can follow the name declaration and any optional attribute blocks. It is a region of regular markup content demarcated by a closing fence `@:@`. With block directives the closing fence has to be placed on an otherwise empty line, with all other directive kinds it can appear anywhere.

Markup example:

```
@:callout(warning)

This is the content of the body.
It is just regular markup.

@:@
```

Combinators:

- `parsedBody`: provides the body content as pre-parsed AST nodes. The type of the body passed to the implementation will be `Seq[Span]`, `Seq[Block]` or `Seq[TemplateSpan]`, depending on the type of the directive.

- `rawBody`: provides the body content as a raw, unparsed `String`.

Combinator Example:

```
val directive = Spans.create("custom") {
  (attribute("name"), parsedBody).mapN { (nameAttribute, bodyContent) =>
    ??? // produce AST span node, bodyContent will be Seq[Span] here
  }
}
```

Access to the Parser

You can request access to the parser of the host language with all extensions the user had installed with the `parser` combinator:

```
(defaultAttribute, parser).mapN { (attrValue, parser) =>
  val parsedSpans = parser("["+attrValue+"]")
  SpanSequence(parsedSpans)
}
```

This is a rare requirement, as in most cases you can just use the `parsedBody` combinator which will give you the entire body segment as a list of AST nodes.

But in scenarios where you need to pre-process or split a raw input string before passing it to the parser, this combinator may come in handy.

Access to the Document Cursor

Finally you can also request access to the document cursor with the `cursor` combinator.

A `DocumentCursor` instance provides access to the project configuration and to all ASTs of the input tree, including those from other documents. It can therefore be used for advanced functionality like producing a table of contents.

Our sample directive implementation showed a complete example of using this combinator in **Access to Configuration**

Link Directives

A link directive is a shortcut for creating span directives that always produce links. It skips some of the boilerplate of regular directives.

An example for a built-in link directive is the `@:api` directive. A link directive always receives exactly one attribute in the form of a string, e.g. `@:api(laika.api.Transformer)`

The directive implementation you need to provide is then merely a simple function (`String, DocumentCursor`) => `SpanLink`, where the String argument is the value of the single attribute.

Example:

```scala
val directive = Links.create("github") { (path, _) =>
  val url = s"https://github.com/our-project/$path"
  SpanLink(Seq(Text(s"GitHub ($path)")), ExternalTarget(url))
}
```

We use the string attribute for creating both, the link text and the URL.

This shortcut can then be used in markup: `@:github(com/example/Hello.py)`. It mostly achieves a reduction in verbosity you'd otherwise need to repeat the same base URL everywhere.

We ignore the second argument, the `DocumentCursor` instance in this case. Like with other directives it gives you access to the configuration and the document AST. You can examine the **Sample Directive** for an implementation that avoids the hard-coding of the base URL with the help of the cursor.

Finally, a special feature of link directives is that they can also be used within native link syntax in such a way that the user can override the default link text while the directive is merely used to construct the URL:

```
Just check our [Hello Example](@:github(com/example/Hello.py))
```

Separator Directives

Body elements can get divided into multiple sub-sections by a special kind of directive, called a separator directive.

For an example from the built-in standard directives, let's have a look at the `@:if` directive:

```
@:if(showSidebar)
<div class="sidebar">...</div>

@:elseIf(showInfobox)
<div class="infobox">...</div>

@:else
<p>This document does not have any sections</p>

@:@
```

The root `@:if` directive is the parent directive in this case, and both the `@:elseIf` and `@:else` directives are separator directives that partition the body. Separator directives are different than normal directives in that they do not need to produce an AST element as they will be passed to the parent directive for processing which then will produce the target AST element.

If you want to see a full example of such a directive, have a look at the implementation of the `@:if` directive in the `StandardDirectives` source.

In this section we'll just show a very small, contrived example. Declaring a separator directive looks just like declaring a normal directive, only that you call `separator` instead of `create`:

```
case class Child (content: Seq[Span])

val sepDir = Spans.separator("child", min = 1) { parsedBody.map(Child) }
```

Here you specify the name of the directive `child`, as well as that it has to be present in the body of the parent directive at least once. Next you use the regular combinators to declare the expected directive parts, in this case only the `parsedBody` that you map to the `Child` type.

Now you can use this directive in the parent:

```
val directive = Spans.create("parent") {
  separatedBody(Seq(sepDir)) map { multipart =>
    val seps = multipart.children.flatMap { sep =>
      Text("Child: ") +: sep.content
    }
    SpanSequence(multipart.mainBody ++ seps)
  }
}
```

You can use the `separatedBody` combinator where you pass all expected child directives (in this case only one) and then map the resulting `Multipart` instance to an AST element. The `Multipart` gives you access to the main body as well as all processed separator directives in `multipart.children`.

This entire directive can then be used like this:

```
@:parent
This is the main body

@:child
This is the body of the separator

@:@
```

Writing Parser Extensions

Implementing a custom parser is one of two possible ways to extend the syntax of a text markup language.

The second option is **Implementing Directives** and in many cases that approach offers more convenience. It is based on a common syntax for declaring a directive and its attributes and body elements. Therefore directives can be implemented without writing a custom parser and without getting familiar with Laika's parser combinators.

There are usually only two scenarios where you may prefer to write a parser extension instead:

- You have very special requirements for the syntax. If, for example, you want to design a new table markup format, this is quite impossible to express with the fixed syntax of a directive.

- You want the most concise syntax for elements that are used frequently. One example is to support short references to issues in a ticket system in the format `#123`. The shortest syntax that is possible with directives would be something like `@:ticket(123)` which you may still find too verbose.

Multi-Pass Markup Parsing

Before delving deeper into the practicalities of writing a parser extension, it is helpful to understand how markup parsing works under the hood in Laika.

In contrast to Laika's HOCON and CSS parsers which are single-pass, markup parsing is always a multi-pass operation.

At the minimum it is parsing in two passes: the first looks for markup which is significant for demarcating a block type (like a blockquote, an ordered list, a code block or a regular paragraph for example). It then further parses the text of each block to look for inline markup.

But it may require any number of additional passes, as blocks or spans themselves can be nested inside each other. An example for a recursive block structure is a list, where each list item may contain nested lists. A span may be recursive where an element like a link allows further inline markup for the link text.

The distinction between block and span parsers is also reflected in the APIs. The document AST comes with base traits `Block` and `Span` (both extending the root `Element` trait) and registration of span and block parsers (which produce `Block` or `Span` AST nodes respectively) is separate and comes with slightly different configuration options as shown in the following sections.

Prerequisites

The content of this chapter builds on top of concepts introduced in other chapters, therefore reading those first might help with following through the examples.

First, all parsers build on top of **Laika's Parser Combinators**. Having its own implementation helps with keeping all functionality tightly integrated and adding some optimizations useful for the specific use case of markup parsing right into the base parsers.

Second, since the parsers we discuss in this chapter all produce AST nodes, it might help to get familiar with **The Document AST** first.

Span Parsers

Span parsers participate in inline parsing. Examples for built-in span parsers are those for emphasized text, literal spans or links.

Adding a span parser to an existing text markup language requires two steps:

- Write the actual implementation of the parser as a type of `PrefixedParser[Span]`.

- Add this declaration to an `ExtensionBundle`.

Let's go through this step by step with an example. We are going to build a simple span parser for a ticket reference in the form of `#123`.

Span Parser Implementation

We require a mandatory `#` symbol followed by one or more digits.

```scala
import laika.ast._
import laika.parse.implicits._

val ticketParser: PrefixedParser[Span] =
  ("#" ~> someOf(CharGroup.digits)).map { num =>
    val url = s"http://our-tracker.com/$num"
    SpanLink(Seq(Text("#" + num)), ExternalTarget(url))
  }
```

- We first look for a literal `"#"`.

- We then require at least one digit. The `someOf` parser covers this as it means "one or more of the specified characters".

- We combine the two parsers with `~>` which means: "concatenate the two parsers, but only keep the result of the right one". See **Laika's Parser Combinators** for details.

- We map the result and create a `SpanLink` node (which implements `Span`). We use the literal input as the link text and then build the URL as an external target.

This first example hard-codes the base URL. If your parser extension is for internal use, this may not be a problem. But if you require a configurable base URL, we later show an enhanced example that has access to the configuration.

Registering a Span Parser

For bundling all your Laika extensions, you need to extend `ExtensionBundle`. In our case we only need to override the `parsers` property and leave everything else at the empty default implementations.

```scala
object TicketSyntax extends ExtensionBundle {

  override val parsers: ParserBundle = ParserBundle(
    spanParsers = Seq(SpanParser.standalone(tickerParser))
  )

}
```

The `SpanParser.standalone` method can be used in cases where your parser does not need access to the parser of the host language for recursive parsing.

Finally you can register your extension together with any built-in extensions you may use:

```scala
laikaExtensions := Seq(
  GitHubFlavor,
  TicketSyntax
)
```

Access to Configuration

This will enhance the previous example by making the base URL configurable.

Access to configuration or other AST nodes is not possible in the parsing phase itself, as each parser executes in isolation and multiple input documents are processed in parallel.

But your parser can return an instance that implements `SpanResolver` instead of directly producing a link. Such a resolver will then participate in the AST transformation phase where its `resolve` method will be invoked:

```scala
case class TicketResolver (num: String, options: Options
 = NoOpt) extends SpanResolver {

  def resolve (cursor: DocumentCursor): Span = {
    cursor.config.get[String]("ticket.baseURL").fold(
      error => InvalidElement(s"Invalid base URL: $error", "#" + num).asSpan,
      baseURL => SpanLink(Seq(Text("#"+num)), ExternalTarget(s"$baseURL$num"))
    )
  }
}
```

The `DocumentCursor` passed to the `resolve` method provides access to the project configuration which we use in this example, but also to all ASTs of the input tree, including other documents. It can therefore be used for advanced functionality like producing a table of contents.

The API of the `cursor.config` property is documented in **Config**.

In our case we expect a string, but we also need to handle errors now, as the access might fail when the value is missing or it's not a string. We return an `InvalidElement` for errors, which is a useful kind of AST node as it allows the user to control the error handling. The presence of such an element will by default cause the transformation to fail with the provided error message shown alongside any other errors encountered. But users can also switch to a "visual debugging" mode by rendering all errors in place. See **Error Handling** for details.

The `options` property of our implementation is a mandatory property of each `Span` or `Block` element. Options are a generic hook that allows users to add an id or multiple styles to a node. It's always best to have an empty default argument like in our example.

With this change in place, the user can now provide the base URL in the builder of the `Transformer`:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue("ticket.baseURL", "https://example.com/issues")
```

The original ticket parser then only needs to be adjusted to return our resolver instead:

```
val ticketParser: PrefixedParser[Span] =
  ("#" ~> someOf(CharGroup.digits)).map(TicketResolver(_))
```

The registration steps are identical to the previous example.

Detecting Markup Boundaries

The examples so far came with a major simplification to keep the code concise. The logic we implemented would also match on a link embedded in markup for example: `http://somwhere.com/page#123`. In such a case we would not want the `#123` substring to be recognized as a ticket link.

For this reason text markup languages usually define a set of rules which define how to detect markup boundaries. In reStructuredText these are defined in a very clear and strict way (**rst-markup-recognition-rules**), while in Markdown the definition is somewhat more fuzzy.

Laika's parser combinators come with convenient helpers to check conditions on preceding and following characters without consuming them. Let's fix our parser implementation:

```
val ticketParser: PrefixedParser[Span] = {
  val letterOrDigit: Char => Boolean = { c =>
    Character.isDigit(c) || Character.isLetter(c)
  }
  val delim = delimiter("#").prevNot(letterOrDigit)
  val ticketNum = someOf(CharGroup.digits)
  val postCond = nextNot(letterOrDigit)

  (delim ~> ticketNum <~ postCond).map(TicketResolver(_))
}
```

Instead of just using a literal parser for the `"#"` we use the `delimiter` parser which comes with convenient shortcuts to check conditions before and after the delimiter.

The end of our span has no explicit delimiter, so we use a standalone `nextNot` condition.

Checking that neither the preceding nor the following character is a letter or digit increases the likeliness we only match on constructs the user actually meant to be ticket references.

Recursive Parsing

In our example we used the `SpanParser.standalone` method for registration. In cases where your parser needs access to the parser of the host language for recursive parsing we need to use the `SpanParser.recursive` entry point instead:

```
SpanParser.recursive { recParsers =>
  ('*' ~> recParsers.recursiveSpans(delimitedBy("*"))).map(Emphasized(_))
}
```

This parser parses markup between two asterisk. It also detects and parses any span kinds provided by either the host markup language or other installed extensions between these two delimiters. The result from the parser obtained with the call to `recursiveSpans` is always of type `Seq[Span]`.

Only this entry point gives you access to a parser that is fully configured with all extensions the user has specified. This parser needs to be injected as you would otherwise hard-code a concrete, fixed set of inline parsers.

Block Parsers

Block parsers participate in parsing block level elements. Examples for built-in block parsers are those for lists, headers, tables or code blocks.

Adding a block parser to an existing text markup language requires two steps:

- Write the actual implementation of the parser as a type of `PrefixedParser[Block]`.

- Add this declaration to an `ExtensionBundle`.

Let's again go through this step by step with an example. We are going to build a block parser for a quoted block. In practice you would not need such an extension, as both supported markup languages in Laika already contain a quoted block element. But it's a good example as the syntax is so simple. In our case we require that each line of such a block starts with the > character:

```
> This line is part of the quotation
> This line, too.
But this line isn't
```

Block Parser Implementation

Let's look at the implementation and examine it line by line:

```scala
import laika.ast._
import laika.parse.implicits._

val quotedBlockParser = BlockParser.recursive { recParsers =>

  val decoratedLine = ">" ~ ws  // '>' followed by whitespace
  val textBlock = BlockParsers.block(decoratedLine, decoratedLine)

  recParsers.recursiveBlocks(textBlock).map(QuotedBlock(_))
}
```

- Our quoted block is a recursive structure, therefore we need to use the corresponding entry point `BlockParser.recursive`. Like the recursive entry point for span parsers, it provides access to the parser of the host language that is fully configured with all extensions the user has specified.

- Next we create the parser for the condition that each line needs to meet to be part of the quoted block. In our case that is the `">"` character, optionally followed by whitespace (the `ws` parser consumes zero or more whitespace characters).

- We then create the parser for the text block based on these predicates. `BlockParsers.block` is a shortcut that will parse lines until a line does not meet the predicate. The result is `List[String]`. See **Base Parsers for Block Elements** below for details about this method.

- Finally we use the recursive parsers we got injected. The call to `recursiveBlocks` "lifts" the specified `Parser[List[String]]` to a `Parser[List[Block]]`

- We map the result and create a `QuotedBlock` node (which implements `Block`). The nested blocks we parsed simply become the children of the quoted block.

Like with span parsers, for blocks which are not recursive you can use the `BlockParser.standalone` entry point.

Registering a Block Parser

For bundling all your Laika extensions, you need to extend `ExtensionBundle`. In our case we only need to override the `parsers` property and leave everything else at the empty default implementations.

```scala
object QuotedBlocks extends ExtensionBundle {

  override val parsers: ParserBundle =
    ParserBundle(blockParsers = Seq(quotedBlockParser))

}
```

Finally you can register your extension together with any built-in extensions you may use:

```
laikaExtensions := Seq(
  GitHubFlavor,
  QuotedBlocks
)
```

Base Parsers for Block Elements

Laika offers a `BlockParsers` object with convenience methods for creating a typical block parser.

One of the most common patterns is parsing a range of lines while removing any decoration that just serves as markup identifying the block type, e.g. the * starting a Markdown list or the > starting a quotation. And then using the parser result to continue parsing recursively, either nested blocks or the inline elements of the block.

For these kind of block elements Laika offers the following method for convenience, which is the utility we used in our example for parsing a quoted block:

```
def block (firstLinePrefix: Parser[Any],
           linePrefix: Parser[Any]): Parser[List[String]]
```

It expects two parsers, one for parsing the prefix of the first line, one for parsing it for all subsequent lines. These parsers may be identical, like in our example for the quoted block. They are both of type `Parser[Any]` as the result will be discarded anyway.

The result of this parser is of type `List[String]` and contains all lines where the specified conditions were met, **minus** the input consumed by the prefix parser. The prefix parsers are not required to consume any input though, if the logic for a particular block does not require stripping off decoration, you can alternatively pass parsers that only check some pre-conditions, but leave all input for the result.

The method above always stops parsing when encountering a blank line on the input, which is common for many types of block elements. For cases where parsing may need to continue beyond blank lines, there is a second overload of this method that allows this:

```
def block (firstLinePrefix: Parser[Any],
           linePrefix: Parser[Any],
           nextBlockPrefix: Parser[Any]): Parser[List[String]]
```

It simply adds a third prefix parser that gets invoked on the beginning of a line following a blank line and is expected to succeed when the line should be treated as a continuation of the block element.

Finally there is a second utility that can be used for indented blocks:

```scala
def indentedBlock (minIndent: Int = 1,
    linePredicate: => Parser[Any] = success(()),
    endsOnBlankLine: Boolean = false,
    firstLineIndented: Boolean = false,
    maxIndent: Int = Int.MaxValue): Parser[String]
```

Like the other utility it allows to specify a few predicates. This method is not used for parsing Markdown's indented blocks, though, as Markdown has a special way of treating whitespace.

Precedence

Both, block and span parsers can specify a precedence:

```scala
BlockParser.recursive { implicit recParsers =>
  ??? // parser impl here
}.withLowPrecedence
```

The default is that extension parsers have higher precedence than host language parsers, which means that they will be tried first and only when they fail the host language parsers will be invoked.

When setting low precedence like in the example above, this is reversed.

Note that precedence only refers to the ordering between the parsers of your extension and those of the host language. The precedence between your extensions, in case you install multiple, is determined by the order you specify them in.

```scala
override val parsers: ParserBundle = ParserBundle(
  spanParsers = Seq(parserA, parserB)
)
```

In this case `parserB` will only be invoked when `parserA` fails. Normally the syntax between inline constructs is different enough that the precedence does not matter. But in some cases extra care is needed. If, for example, you provide parsers for spans between double asterisk `**` and between single asterisk `*`, the former must be specified first, as otherwise the single asterisk parser would 100% shadow the double one and consume all matching input itself.

Internal Design & Performance

For span parsers the parser you are passing to the `standalone` or `recursive` methods must be of type `PrefixedParser[Span]`, which is a subtype of the base trait `Parser[T]`.

This is a trait implemented by all parsers with a stable prefix that can be optimized, like literal string parsers or many other base parsers like the `someOf` or `delimiter` parsers. In most cases this is nothing you need to worry about, as most of the building blocks you use already implement this trait. In the rare occasions

where this is not the case, you might need to wrap your inline parser in a custom implementation of the `PrefixedParser` trait.

This restriction is necessary as inline parsers cause checks to be performed on each character. This is therefore the key hot spot for performance optimizations. For details see **Performance Optimizations** in the chapter about Laika's parser combinators.

For block parsers this restriction is lifted, you can pass a regular `Parser[Block]` to the registration entry points. One reason is that optimizing block parsers is much less critical, as they are only invoked (more or less) after a blank line has been read, not on each input character. Another reason is that the nature of some block elements makes this type of optimization impractical. An underlines header for example does not have any concrete set of start characters.

But even though it is not required, if you do actually pass `PrefixedParser[Block]` it will nevertheless get optimized under the hood, just that the impact of this optimization will be smaller than for span parsers.

Adding Syntax Highlighters

Laika has its own built-in syntax highlighters (based on its parser combinators).

This manual itself is a showcase for this functionality. All code samples shown are highlighted by Laika's own syntax support.

This chapter explains the general design goals as well as the basic building blocks for adding your own highlighters.

Design Goals

- **Everything is pluggable and customizable.**

  This includes making it easy to add support for additional languages as well as ignoring the built-in highlighter altogether and use external tools like `highlight.js` instead.

- **The mechanism is not tied to HTML output.**

  The analyzed code is part of the document AST and can be processed and transformed like other AST nodes before rendering, making it a part of the common processing pipeline and not a bolt-on step after rendering, that would require the availability of JavaScript for the rendered output for example.

  This way the highlighted nodes can be rendered in any output format like EPUB, PDF or any 3rd-party format.

- **Focus on display comfort over validation.**

  It's primary use case is documentation or blogs, where the highlighter is supposed to improve the readability of the code. It is not designed to serve as a basis for code editors or include error highlighting and messages.

  Therefore it is also beyond the scope of this utility to detect larger constructs like entire method signatures. In some languages like Java this would come with a lot of ambiguity as a method declaration does not start with a keyword like `def`. It would require to get much closer to having a full blown lexer and analyzer for the language than this utility is designed for.

  As a consequence its syntax definitions are more similar in scope and precision to those of lightweight tools like `highlight.js` than to the lower-level nature of sublime-text syntax definitions for example. If validation is required the recommended approach in Laika is to rely on other tools like MDoc for Scala validation and then only use Laika for highlighting the result.

- **Avoiding the regular expression mazes of other tools.**

  It offers a lot of higher level building blocks for assembling highlighters for common constructs like string or number literals which are often quite similar between languages. The definitions then become more of a declarative list of supported constructs.

But even in cases where a custom parser is required, it encourages the use of Laika's built-in parser combinator framework, which is a more composable and type-safe way to define a parser than a cryptic (and stringly) regular expression.

None of the built-in parsers for text markup, HOCON, CSS or syntax highlighting use any regular expressions.

Prerequisites

The content of this chapter builds on top of concepts introduced in other chapters.

First, even though Laika's highlighting support defines a lot of higher level building blocks that allow for a mere declarative assembling of the supported constructs for a language, some additions of custom parsers might be necessary, depending on the complexity of the target language. In that case it's recommended to use **Laika's Parser Combinators**.

Second, highlighters produce a list of `CodeSpan` nodes which is one node type of Laika's document AST. Even though this is usually the only node type you deal with, it might help to get familiar with **The Document AST** first.

The SyntaxHighlighter Trait

These are the two abstract methods a new highlighter needs to implement:

```scala
def language: NonEmptyList[String]

def spanParsers: Seq[CodeSpanParser]
```

- The `language` property holds one or more names for the language that will be used to identify the language when syntax like fenced code blocks are used.

  Concretely this means that if you specify `python` and `py` as the names for example, the implementation will be used for any fenced code block that starts with either ```py or ```python.

  It's not tied to Markdown syntax though. reStructuredText has its own way of specifying a code block with language identifier, and the highlighter will be used for those, too.

- The `spanParsers` property holds the actual parsers. Like with markup parsers they can be defined independently and then simply get listed in this trait.

Let's start by looking at how you can assemble these span parsers by using some of the building blocks for common language constructs.

We later look at how to add other constructs based on custom parsers.

Using Existing Building Blocks

The building blocks shown in this section are meant to be used as shortcuts. Whenever they do not provide the exact syntax option that you need, you can always fall back to **Defining Custom Parsers**.

Our code examples in this section define highlighters for a Scala-ish language, but omit many details for the sake of brevity. You can examine the source code of Laika's built-in highlighters for more complete examples.

String Literals

There are shortcuts for defining single- and multi-line strings together with escape sequences:

```scala
val spanParsers: Seq[CodeSpanParser] = Seq(
  StringLiteral.multiLine("\"\"\""),
  StringLiteral.singleLine('"').embed(
    StringLiteral.Escape.unicode ++ StringLiteral.Escape.char
  )
)
```

The `multiline` and `singleLine` builders both require to specify the delimiter. When the start and end delimiter are identical you can use the single-argument overload.

For a single line string we also define valid escape sequences which will be used to detect nested spans. These are existing building blocks, too.

There are additional escape options like octal and hex, as well as building blocks to define substitutions, e.g. Scala's `s"some $ref"` syntax.

Character Literals

Character literals have fewer options than string literals, but are otherwise quite similar:

```scala
val spanParsers: Seq[CodeSpanParser] = Seq(
  CharLiteral.standard.embed(
    StringLiteral.Escape.unicode ++ StringLiteral.Escape.char
  )
)
```

The `standard` builder uses single quotes as delimiters, but you can specify a different one if required. Like with our string literal example, we define two kinds of escapes that can occur in a character literal.

Numeric Literals

There are shortcuts for the most common types of decimal, hex, octal and binary number literals:

```scala
val spanParsers: Seq[CodeSpanParser] = Seq(
  NumberLiteral.hex
    .withUnderscores
    .withSuffix(NumericSuffix.long),
  NumberLiteral.decimalInt
    .withUnderscores
    .withSuffix(NumericSuffix.long | NumericSuffix.float)
)
```

Here we are defining two types of number literals: hexadecimal and decimal integers.

In both cases we allow the use of underscores as separators (e.g. `1,000,000`).

We also define which kinds of type indicators we allow as a suffix and again use existing parsers for widely used syntax (`NumericSuffix.long` parses `L` or `l` for example).

Identifiers

This is how the identifier category is defined for the Scala highlighter:

```scala
val identifier = Identifier.alphaNum
  .withIdStartChars('_','$')
  .withCategoryChooser(Identifier.upperCaseTypeName)
```

It expects any alphanumerical identifier that is also allowed to have _ or $ as a start character.

We then pass a function as a "category chooser" that assigns a category based on the parsed identifier. It is a very simple function that serves as a relatively useful approximation: it picks `CodeCategory.TypeName` when the identifier starts with an uppercase letter, otherwise it assigns `CodeCategory.Identifier`.

This is obviously not always correct, but good enough for most use cases. As discussed in the design goals, this highlighting utility does not aim to get close to defining full lexers and parsers, and assigning the category correctly in all cases would require full context of the parsed construct. Many highlighters you might be using with other tools are equally pragmatic.

Keywords

Keywords can simply be listed as string literals:

```scala
val spanParsers: Seq[CodeSpanParser] = Seq(
  Keywords("abstract", "case", "catch", "class", "def")
)
```

It detects word boundaries so that it will not match on sub-strings.

By default the parser will assign `CodeCategory.Keyword` to any matching string. If you want to use literal matches, but with a different code category, there is an alternative constructor:

```
Keywords(CodeCategory.BooleanLiteral)("true", "false")
```

Comments

Shortcuts exist for defining single- and multi-line comments:

```
val spanParsers: Seq[CodeSpanParser] = Seq(
  Comment.singleLine("//"),
  Comment.multiLine("/*", "*/")
)
```

The code should be self-explanatory here, we simply define the delimiters for the comment syntax of our language. The single-line variant terminates at the end of the line.

Defining Custom Parsers

The basic building blocks presented in the previous section help a lot with reducing boilerplate, but in most cases you end up with the need to have at least a handful of hand-written parsers, too.

For a little example, let's implement a highlighter for Scala's backtick identifiers (e.g. `` `tag-name` ``) which is quite straightforward, but not included in the reusable builders since it's not a very common construct:

```
import laika.parse.builders._
import laika.parse.implicits._

val backtickId: CodeSpanParser = CodeSpanParser(CodeCategory.Identifier) {
    (oneOf('`') ~ anyNot('\n', '`') ~ oneOf('`')).source
  }
```

In the first line of the definition we specify the category that should be assigned to the result of the parser.

The next line is a standard parser definition using some of the building blocks of **Laika's Parser Combinators**. We simply expect a backtick as the start and end delimiter and allow anything in-between apart from newline or backtick characters.

The call to `.source` is a convenient shortcut that discards the result of the three parsers and instead provides the source of the entire input by those three parsers as the result. This is often preferable when all we would do with the three results would be concatenate them back to one string.

The code parser we now defined can then be registered in our `spanParsers` sequence like in all other examples.

The entry point above is just one option. It's only appropriate if we provide a string parser and want the same category assigned to the entire string.

There is an alternative factory method, where we specify a `PrefixedParser[Seq[CodeSpan]]` where we have full control over how we slice the input into multiple categories.

Embedding other Languages

In some cases a language or format may have regions that require a highlighter for a different syntax. A classic example is HTML which may have CSS syntax inside a `<style>` tag and JavaScript syntax inside a `<script>` tag.

Laika comes with a parser builder that helps with the creation of such a parser.

Let's pick CSS in HTML as an example and show a simplified definition for a such a style tag:

```
val styleTagParser: PrefixedParser[Seq[CodeSpan]] = {
  val cat = CodeCategory.Tag.Name
  val bodyAndEndTag =
    EmbeddedCodeSpans.parser(delimitedBy("</style>"), CSSSyntax)

  (literal("<style>") ~> bodyAndEndTag).map { css =>
    CodeSpan("<style>", cat) +: css :+ CodeSpan("</style>", cat)
  }
```

The key here is the third line. We define a region to parse with `delimitedBy("</style>")` which consumes text until it sees the specified end delimiter. See **Delimiters** for details. And we specify the syntax to switch to until we reach that delimiter, in this case the existing `CSSSyntax`

The result (the `css` param) is of type `Seq[CodeSpan]`, containing the spans detected by the CSS highlighter. We then wrap the code spans for the start tag and end tag around it for the final result.

One of the simplifications to keep the example simple is that we assign the category `CodeCategory.Tag.Name` to the entire tag, including the angle brackets. Normally you would keep them separate under `CodeCategory.Tag.Punctuation`.

Parser Precedence

The precedence rules are identical to those for declaring markup parsers.

The parser precedence is determined by the order you specify them in. This means they will be "tried" on input in that exact order. The second parser in the list will only be invoked on a particular input when the first fails, and so on.

This is the logical execution model only, the exact runtime behaviour may differ due to performance optimizations, but without breaking the guarantees of respecting the order you specify.

In many cases the difference in the syntax between constructs is high enough that the precedence does not matter. But in some cases extra care is needed.

If, for example, you provide a parser for a span delimited by one or two occurrences of the same character, the two-character version must be specified first, as otherwise the parser with the single-character delimiter would 100% shadow the double one and consume all matching input itself, unless it contains a guard against it.

Similarly, parsers like those for identifiers which do not have a limited set of start delimiters usually need to occur near the bottom of the list of parsers you register.

Registering a Highlighter

First you have to assemble all the parsers in a `SyntaxHighlighter` implementation:

```scala
object FooHighlighter extends SyntaxHighlighter {

  val language: NonEmptyList[String] = NonEmptyList.one("foo")

  val spanParsers: Seq[CodeSpanParser] = Seq(
    // all your definitions here
  )

}
```

Finally, like all other types of extensions, the highlighter needs to be registered with an `ExtensionBundle`:

```scala
case object MyExtensions extends ExtensionBundle {

  override def parsers: ParserBundle = ParserBundle(
    syntaxHighlighters = Seq(
      FooHighlighter
    )
  )
}
```

You can bundle multiple highlighters in a single instance.

Finally you can register your extension together with any built-in extensions you may use:

```scala
laikaExtensions := Seq(
  GitHubFlavor,
  MyExtensions
)
```

New Markup or Output Formats

All the other chapters in the "Extending Laika" section of the manual deal with a customization option that at some point becomes part of an `ExtensionBundle`. As **Anatomy of the API** showed, it is one of the major API hooks for building a transformer:

```
val transformer = Transformer
  .from(Markdown)                     implements
                                      MarkupFormat
  .to(HTML)
                                      implements
                                      RenderFormat
  .using(
    GitHubFlavor,
                                      implements
    SyntaxHighlighting                ExtensionBundle
  )
  .build
```
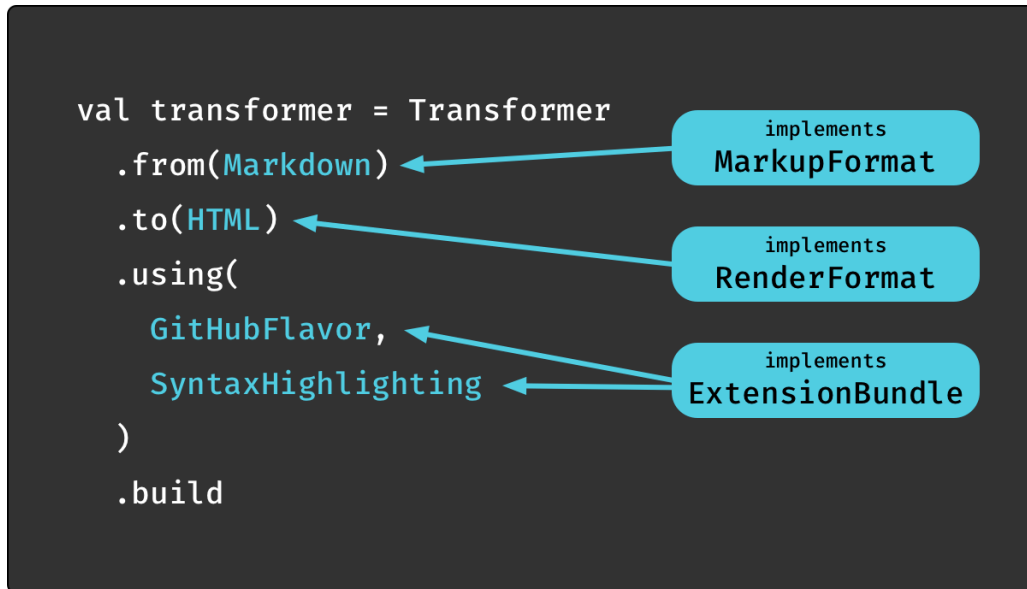
While an `ExtensionBundle` is about enhancing functionality for existing input and output formats, this chapter finally is about actually adding new formats. Therefore it deals with the other two types in the diagram: `MarkupFormat` and `RenderFormat[FMT]`.

Implementing a Markup Format

Laika currently supports two markup formats out of the box: Markdown and reStructuredText. Having two formats from the beginning greatly helped in shaping a document model that is not tied to the specifics of a particular text markup language.

Making it as straightforward as possible to add support for more formats like ASCIIDoc or Textile was one of Laika's initial design goals.

Parser Prerequisites

The content of this chapter builds on top of concepts introduced in other chapters, therefore it's recommended to read those first.

First, all parsers build on top of **Laika's Parser Combinators**. Having its own implementation helps with keeping all functionality tightly integrated and adding some optimizations useful for the specific use case of markup parsing right into the base parsers.

Second, since block and span parsers that together provide the markup implementation all produce AST nodes, it might help to get familiar with **The Document AST** first.

Finally, **Writing Parser Extensions** covers a lot of ground relevant for adding a new format, too. It walks through the sample implementation of a span parser.

The main difference is merely that a block or span parser serving as an extension for existing markup languages will be registered with an `ExtensionBundle`, while the parsers for a new format need to be registered in a `MarkupFormat`.

The MarkupFormat Trait

The contract a markup implementation has to adhere to is captured in the following trait:

```scala
trait MarkupFormat {

  def fileSuffixes: Set[String]

  def blockParsers: Seq[BlockParserBuilder]

  def spanParsers: Seq[SpanParserBuilder]

  def extensions: Seq[ExtensionBundle]

}
```

These are the four abstract method each parser has to implement.

- The `fileSuffixes` method should simply return the set of supported file suffixes (without the '.'). For Markdown this would be `Set("md", "markdown")` for example. It is not recommended to support generic suffixes like `txt` as this could lead to conflicts with other parsers.

  Laika supports a setup where input directories may contain source files written in different markup languages. In this case the format detection is solely based on the file suffix.

- The `blockParsers` and `spanParsers` collections provide the definitions for the actual markup parsers. These are the same types you've already seen when registering parsers as extensions.

  It may sound too good to be true, but the task of assembling the building blocks of a markup language in Laika is indeed merely declarative. The parsers for individual elements like lists or links can almost always be implemented in isolation with little or no interdependencies.

  Writing Parser Extensions contains a walk-through of a span parser implementation. You can also browse the code for Laika's Markdown and reStructuredText support for more examples.

- The `extensions` collection allows to add functionality beyond just markup parsing. It uses the same `ExtensionBundle` trait that exists for users to customize a transformation.

  Whenever a format that defines extensions is used, they are merged with the user-supplied extensions. This collection may very well remain empty for some formats.

Finally there are three concrete methods that may be overridden if required:

```scala
def description: String = toString

def escapedChar: Parser[String] = TextParsers.oneChar

def createBlockListParser (parser: Parser[Block]): Parser[Seq[Block]] =
  (parser <~ opt(blankLines)).rep
```

- The `description` method allows to add details for tooling and logging.

- The `escapedChar` method controls the parsing of an escape sequence (the character after a backslash). The default implementation accepts any character.

- Finally, `createBlockListParser` controls how a parser for a single block is turned into a parser for a sequence of blocks. The default implementation just skips blank lines between the blocks and repeats the same parser. The parser passed to this method already includes all the (merged) block parsers this trait has specified as well as any extensions a user might have installed.

Parser Precedence

The parser precedence is determined by the order you specify them in. This means they will be "tried" on input in that exact order. The second parser in the list will only be invoked on a particular input when the first fails, and so on.

This is the logical execution model only, the exact runtime behaviour may differ due to performance optimizations, but without breaking the guarantees of respecting the order you specify.

Normally the difference in the syntax between markup constructs is high enough that the precedence does not matter. But in some cases extra care is needed. If, for example, you provide parsers for spans between double asterisk `**` and between single asterisk `*`, the former must be specified first, as otherwise the single asterisk parser would 100% shadow the double one and consume all matching input itself, unless it contains a guard against it.

Implementing a Render Format

Laika currently supports several output formats out of the box: HTML, EPUB, PDF, XSL-FO and AST. XSL-FO mostly serves as an interim format for PDF output, but can also be used as the final target format. AST is a renderer that provides a formatted output of the document AST for debugging purposes.

Making it as straightforward as possible to add support for additional output formats, potentially as a 3rd-party library, was one of Laika's initial design goals.

Renderer Prerequisites

The content of this chapter builds on top of concepts introduced in other chapters, therefore it's recommended to read those first.

First, since renderers have to pattern match on AST nodes the engine passes over, it might help to get familiar with **The Document AST** first.

Second, **Overriding Renderers** shows examples for how to override the renderer of a particular output format for one or more specific AST node types only.

The main difference is that a renderer serving as an override for existing output formats will be registered with an `ExtensionBundle`, while the renderers for a new format need to be registered in a `RenderFormat`.

The second difference is that a `RenderFormat` naturally has to deal with all potential AST nodes, not just a subset like an override.

The RenderFormat Trait

A renderer has to implement the following trait:

```scala
trait RenderFormat[FMT] {

  def fileSuffix: String

  def defaultRenderer: (FMT, Element) => String

  def formatterFactory: RenderContext[FMT] => FMT

}
```

- The `fileSuffix` method provides the suffix to append when writing files in this format (without the ".").

- The `defaultRenderer` represents the actual renderer. It takes both, a formatter instance and the element to render and returns a String in the target format.

- `formatterFactory` is the formatter instance for the target format. A new instance of this formatter gets created for each render operation.

- The `RenderContext` passed to the factory function contains the root element of the AST and the delegate render function which your formatter is supposed to use for rendering children. You need to use this indirection as the provided delegate contains the render overrides the user might have installed which your default render implementation cannot be aware of.

- `FMT` is a type parameter representing the Formatter API specific to the output format. For the built-in renderers, this is `FOFormatter` for `PDF`, `HTMLFormatter` for `HTML` and `EPUB` and finally `TextFormatter` for the `AST` renderer.

The Render Function

This `defaultRenderer` function should usually adhere to these rules:

- When given an element that is a container type that contains child elements (like `Paragraph` or `BulletList`), it should never render the children itself, but instead delegate to the Formatter API, so that user-defined render overrides can kick in for individual element types.

- It should expect unknown element types. Since parsers can also be extended, the document tree can contain nodes which are not part of the default node types provided by Laika. The root trait `Element` is *not* sealed on purpose.

  Dealing with unknown types is best achieved by not only matching on concrete types, but also on some of the base traits.

  If, for example, you already handled all the known `SpanContainers`, like `Paragraph`, `Header` or `CodeBlock`, you would ideally add a fallback pattern that matches on any `SpanContainer` and provides a default fallback. This way, even though you don't know how to best render this particular container, at least all its children will most likely get rendered correctly.

  As an absolute worst-case fallback it should add a catch-all pattern that returns an empty string.

Let's look at a minimal excerpt of a hypothetical HTML render function:

```scala
def renderElement (fmt: HTMLFormatter, elem: Element): String = {

  elem match {
    case Paragraph(content,opt) => fmt.element("p", opt, content)

    case Emphasized(content,opt) => fmt.element("em", opt, content)

    /* [other cases ...] */

    /* [fallbacks for unknown elements] */
  }
}
```

As you see, the function never deals with children (the `content` attribute of many node types) directly. Instead it passes them to the Formatter API which delegates to the composed render function. This way user-specified render overrides can kick in on every step of the recursion.

In the context of HTML it means that in most cases your implementation renders one tag only before delegating, in the example above those are `<p>` and `<em>` tags respectively.

Choosing a Formatter API

Depending on the target format your renderer may use the `TextFormatter` or `HTMLFormatter` APIs, which are explained in **The Formatter APIs**.

Alternatively it may create its own API, but you should keep in mind then, that this API will also get used by users overriding renderers for specific nodes. Therefore it should be convenient and straightforward to use and well documented (e.g. full scaladoc).

Even when creating your own formatter it's probably most convenient to at least extend `TextFormatter`, which contains base logic for indentation and delegating to child renderers.

Costs of Avoiding Side Effects

As you have seen the render function returns a `String` value, which the engine will then build up recursively to represent the final output. It can therefore get implemented as a pure function, fully referentially transparent.

Earlier Laika releases had a different API which was side-effecting and returning Unit. Renderers directly wrote to the output stream, only hidden behind a generic, side-effecting delegate API.

Version 0.12 in 2019 then introduced full referential transparency and one of the necessary changes was the change of the render function signature. These changes (taken together, not specifically that for the Render API) caused a performance drop of roughly 10%. It felt reasonable to accept this cost given how much it cleaned up the API and how it lifted the library to meet expectations of developers who prefer a purely functional programming style.

The decent performance of Laika stems mostly from a few radical optimization on the parser side, which led to much better performance compared to some older combinator-based Markdown parsers.

The alternative would have been to build on top of a functional streaming API like fs2, as this might have preserved both, the old performance characteristics as well as full referential transparency. But it still would have complicated the API and introduced a dependency that is not needed anywhere else in the `laika-core` module, which does not even require `cats-effect`.

Laika's Parser Combinators

Since the 0.8 release in 2018 Laika comes with its own parser combinator library.

The decision was based on the goal to find the sweet spot between ease of use, flexibility and performance. It's to a large degree a general purpose parser library for character input, but also has a few aspects tailored for Laika's main requirements: parsing of text markup which is a multi-pass process and providing a maximum of flexibility for user provided extensions of the markup syntax.

The library is currently part of the `laika-core` artifact, but has the potential to become a separate micro-lib, if some users voice an interest in using it without Laika's other features.

The Parser Trait

A parser instance is invoked at a particular offset into an input and performs the following tasks:

1) produce a result or fail with a message

2) optionally consume some of the input

This is encapsulated in the single abstract method of the `Parser` trait:

```scala
trait Parser[T] {
  def parse (in: ParserContext): Parsed[T]
}
```

The `ParserContext` contains an API for reading from the input at the current offset and for capturing or consuming some of the input.

The returned result of type `Parsed[T]` is a little ADT with the following types:

```scala
case class Success[+T] (result: T, next: ParserContext) extends Parsed[T]

case class Failure (msg: Message, next: ParserContext) extends Parsed[Nothing]
```

In case of success the result will be returned alongside a new `ParserContext` that may have consumed some of the input. Consuming input is optional as some parsers only look ahead on the input to check a precondition.

In case of an error the returned value will contain a (lazily constructed) error message and a `ParserContext` which will usually be at the same offset as the context passed in, so that subsequent parsers can try to read from the same location.

The trait is mostly shown for providing some background about the most basic building block of the combinator library. You will rarely create an implementation of `Parser` yourself as the existing building blocks cover a lot of scenarios already.

Parsing Text

One of the basic building blocks for parsers are the text parsers provided by the library.

They deliberately do not integrate with or otherwise use regular expressions and prefer a combinator DSL instead. While being significantly more verbose, combinators are a more type-safe, composable and also often more legible way to define text parsers, in particular with larger, more complex expressions.

For defining parsers you need the following two imports:

```
import laika.parse.builders._
import laika.parse.implicits._
```

The following section demonstrate some of the most commonly used text parsers. For the full API see **TextParsers$**.

Literal Text

A literal parser only matches on an exactly identical input and is often used for parsing keywords or delimiters consisting of one or more characters.

`literal("class")` for example only matches on the `class` keyword.

In many cases declaring the literal parser explicitly can be skipped. The imports shown above contain extension methods that allow the use of raw strings where otherwise a `Parser` type is expected.

`"[" ~ anyNot(']') ~ "]"` is a shortcut for `literal("[") ~ anyNot(']') ~ literal("]")`.

Character Groups

To parse a range of characters you can either provide a set of matching characters or a predicate:

- `anyOf('*', '+', '-')` reads zero or more characters that match any of the three specified symbols. E.g. reading `**--**..` would provide the result `**--**`.

- `anyNot('*', '+', '-')` is the opposite, it reads zero or more characters that do not match the specified symbols. E.g. reading `..,,**--**` would provide the result `..,,`.

- `anyWhile(_.isUpper)` reads zero or more upper case characters. Any predicate `Char => Boolean` can be specified.

- The `range` helper constructs set of characters, e.g. `anyOf(range('a','z'))`.

- There are predefined character groups for common scenarios, e.g. `anyOf(CharGroup.hexDigit)`.

For all the shown parsers the result may be empty and the parser will always succeed. There are ways to set conditions on the length of the input shown in the next section.

Length Constraints

All character parsers come with `min`, `max` and `take` methods to specify constraints on the length of the result.

- `anyOf(CharGroup.alphaNum).min(3)` expects 3 or more matching characters. It fails when it matches on fewer than that.

- `anyOf(CharGroup.alphaNum).min(3)` expects 0 to 3 characters. It always succeeds. After 3 characters are read it simply ignores any further matching input.

- `anyOf(CharGroup.alphaNum).take(3)` is a shortcut for `anyOf(CharGroup.alphaNum).min(3).max(3)`.

- `someOf`, `someNot` and `someWhile` builders are shortcuts for `anyOf`, `anyNot` and `anyWhile` that expect at least one matching character. E.g. `someOf(CharGroup.alphaNum)` is equivalent to `anyOf(CharGroup.alphaNum).min(1)`. That's not that many characters saved, but used so frequently that it's worthwhile.

- Likewise `oneOf`, `oneNot` and `oneIf` are shortcuts for parsers that parse exactly one character. E.g. `oneOf(CharGroup.alphaNum)` is equivalent to `anyOf(CharGroup.alphaNum).take(1)`.

- The `count` method creates a parser that provides the number of characters read as the result. Commonly used for parsing decoration where the length is significant, e.g. for Markdown headers starting with one or more #: `someOf('#').count`.

Combinators

So far we were only parsing a single range of input based a condition or set of accepted input characters. In every kind of real-world scenario you would need to combine these low-level parsers to larger constructs.

Concatenation

The ~ combines the result of two parsers and only succeeds if both of them succeed:

```
val p = oneOf('$','_') ~ someOf(range('a', 'z'))
```

The above parser expect exactly one occurrence of either $ or _, followed by one or more occurrences of a lowercase letter.

The result will be `String ~ String`, where ~ is a case class that allows to map on the result with the same symbol:

```
p.map { case firstChar ~ lowerCaseLetter => Seq(firstChar, lowerCaseLetter) }
```

In many cases we are only interested in one of the results of a concatenation, when some of the results are known for example. The ~> combinator ignores the left result, <~ ignores the right one:

```
val p = "<" ~> someOf(range('a', 'z')) <~ ">"
```

The result of this parser will be `String` as the first and last result will be ignored.

Alternatives

Two parsers can be defined to be tried as alternatives, where the second will only be invoked if the first parser fails:

```
("\"" ~> anyBut('"') <~ "\"") | someBut(' ')
```

The example above parses either text enclosed in double quotes or a string without spaces. The type of the result will be the lowest upper bound of the individual results (like with `Option.orElse`).

The resulting parser will succeed if either the first or the second parser succeeds. If both of them fail, the returned error will be the one from the parser which successfully read the most characters. This will of course also apply to longer chains of alternatives like `a | b | c | d`. In many cases this will be the error message that is the most helpful for the user.

Repetitions

The same parser can be invoked repeatedly, while collecting all individual results on the way:

```
someOf(CharGroup.alphaNum) ~ (',' ~> someOf(CharGroup.alphaNum).rep)
```

The above reads a non-empty sequence of alphanumerical characters followed by zero or more repetitions of a comma followed by another sequence of characters. The result will be `String ~ Seq[String]`

This pattern is so common that there is also a shortcut for repeating with a separator:

```
someOf(CharGroup.alphaNum).rep(',')
```

This parser's behaviour is identical to the previous one.

The number of repetitions can be further constrained:

```
someOf(CharGroup.hexDigit).max(4)
```

The above reads between 1 to 4 hexadecimal digits.

When using the `min` constraint the parser will fail when it does not reach the specified minimum number of repetitions, while the `max` constraint will always succeed and simply ignore subsequent repetitions.

There is also a shortcut called `.take(4)` which is identical to `.min(4).max(4)`.

Producing Results

All of our preceding examples defined lower-level text parsers that produce a `String` result or `String ~ String` in case of concatenation and `Seq[String]` in case of repetition.

In many real-world scenarios parsers are used to build up a form of AST of the parsed input. All concrete parsers in Laika do this, the ones of text markup, HOCON and CSS all produce a model representing the input.

First there is the classic map:

```
def map[U] (f: T => U): Parser[U]
```

The API also offers methods that allow to check pre-conditions on the result, potentially causing the parser to fail if they are not met:

```
def evalMap [U] (f: T => Either[String, U]): Parser[U]
def collect [U] (f: PartialFunction[T, U]): Parser[U]
```

While `evalMap` maps to an `Either` where a `Left` result will cause the parser to fail, `collect` applies a partial function and causes the parser to fail if the function is not defined for the result.

```
someOf(CharGroup.digit).evalMap { res =>
  val num = res.toInt
  Either.cond(num % 3 == 0, num, "Number must be divisible by three")
}
```

The example above creates a parser that reads any number divisible by 3 or fails otherwise.

You can also chain parsers with `flatMap`. The example parses a start delimiter out of 3 options and then looks for a matching delimiter to close the span:

```
oneOf('*', '-', '+').flatMap { res =>
  someBut(res.charAt(0)) <~ literal(res)
}
```

The second parser will receive the result of the first parser and will continue parsing on the input left over by the first.

You can also ignore the original result of a parser and hard-code a result that should be used if the parser succeeds:

```
case object Fence

literal("```").as(Fence)
```

Another option is to ignore the results of a concatenation and instead use the entire consumed input as a result:

```
("\"" ~> anyBut('"') <~ "\"").source
```

This is usually more convenient in all cases where the result you would produce would have been the string concatenation of the individual results anyway. Laika's own parsers use this very frequently.

Finally, while all the methods shown so far are available for all kinds of `Parser[T]`, the `laika.parse.implicits._` import also provides a few convenient shortcuts for parsers of a certain result type.

A result of concatenating a single result with a repetition can be combined into a single list with `concat` for example:

```
someOf(CharGroup.alphaNum) ~ (',' ~> someOf(CharGroup.alphaNum).rep).concat
```

This will turn a parser with the result `String ~ Seq[String]` into a parser for `Seq[String]`.

There are similar variants for results of type `Seq[T] ~ Seq[T]` of various arities.

Delimiters

In many cases the task of a parser is not only to verify whether the input at the current position matches a particular predicate, but also whether the surrounding characters meet certain criteria.

Let's take the parsing of an emphasized span in text markup as an example. Most languages define exceptions where the span `*here*` is not parsed as emphasized, for example when it appears in the middle of a word like `in*this*example`.

Laika's parser combinators come with convenient helpers to check conditions on preceding and following characters without consuming them. Doing this manually with existing low-level combinators is possible, but much more verbose and error-prone.

A simplified definition of such a parser for an emphasized span could look like this:

```
val letterOrDigit: Char => Boolean = { c =>
  Character.isDigit(c) || Character.isLetter(c)
}
val start = delimiter("*").prevNot(letterOrDigit)
val end   = delimiter("*").nextNot(letterOrDigit)

start ~> someNot('*') <~ end
```

Instead of just using a literal parser for the `"*"` we use the `delimiter` parser and its integrated shortcuts to check conditions before and after the delimiter.

Instead of just parsing the delimiter itself, an even more powerful variant parses a span of text until a specified delimiter is seen.

The last line in the previous example can be shortened to:

```
start ~> delimitedBy(end)
```

But this variant does not only save a bit of typing, it also comes with additional functionality for defining the parsed span. One of them is the option to specify conditions under which the parser should fail, even if it would later see the given delimiter.

Let's assume we want an emphasized span that is not allowed to span multiple lines. We can express this with `failOn`:

```
start ~> delimitedBy(end).failOn('\n')
```

Other methods are `acceptEOF` which tells the parser to succeed if either the specified delimiter or the end of the input is reached or `nonEmpty` which causes the parser to fail if the delimiter is seen before consuming any input.

Parsing Nested Spans

The last feature area of the parser combinators we are going to look at is the recursive parsing of nested spans, which is at the heart of inline text markup parsing and another area where the library takes away a lot of the boilerplate you'd have to write with simple low-level combinators.

For a real world example, let's assume we want to parse markup that allows a Markdown-like link syntax, between brackets, with potential markup inside. We don't allow this syntax to span multiple lines.

This is how a simplified implementation could look like:

```
val nestedSpanParsers: Seq[PrefixedParser[Span]] = ???
val linkSpanParser = delimitedBy("]").failOn('\n')

"[" ~> InlineParsers.spans(linkSpanParser).embedAll(nestedSpanParsers)
```

We skip the definitions of all the nested parsers, with typical markup parsing this would be a handful or a dozen. In the second line we define the parser for the link span itself, using the `delimitedBy` builder which we introduced in the previous section.

Finally we pass this parser to the `InlineParsers.spans` constructor and then tell it to embed all the nested span parsers we defined.

The resulting parser will:

- Fail when it sees a newline before seeing the end delimiter.

- Suspend parsing of this span when a pattern signalling the start of a nested span is seen. It will then parse the nested span, add it to the result and resume parsing of the top span.

- It will finally succeed and provide the result (as a `Seq[Span]`) when it sees the end delimiter.

There is an alternative constructor `InlineParsers.text` where the result is `String` and not `Seq[Span]`. It is less often used, but comes in handy when you want to support something like text between parenthesis that should not stop when a closing paren of a nested pair is seen.

The example above is applicable for a scenario where the author of the code has full control over the kind of nested spans which are allowed. In Laika's text markup support this is not the case, as the user can install markup extensions. For this scenario the nested parsers need to be injected from the environment, which is demonstrated in **Recursive Parsing** in the chapter on writing parser extensions for markup.

Performance Optimizations

The final example in the previous section demonstrated the principles behind parsing inline markup. These kind of parsers are quite challenging to optimize as they are at a hotspot of markup parsing and recurse between user-provided and library-provided parsers which makes it harder to optimize based on static analysis.

Inline Parsing Challenges

Laika has therefore chosen an approach where the optimization happens at parser construction time, at a point where all participating parsers are known.

The key point of the optimization is to avoid a naive combinator-based approach for logic that has to execute for nearly every character of the input stream. A naive approach would try something like `spanA | spanB | spanC | spanD | plainText` on each character, where `spanA` end so on are the supported inline constructs. It would need to try a long chain of parsers where in probably 95% of cases it would need to fall back to the last one: read the input as plain text, the most likely outcome.

On top of the long list of choices, the parser has to look for other conditions, too, as shown in the previous section: characters that cause the current span to fail and those who terminate it.

Laika combines all these aspects into a single quick check, a simple array lookup that first determines whether the current input character is an "interesting" one or just a character to consume and move on. Only when this condition is met, the actual parsers are invoked.

The PrefixedParser trait

To facilitate this at runtime, the library comes with an optimizable sub-trait of `Parser[T]` called `PrefixedParser[T]`. On top of the actual parsing logic it encapsulates a separate condition for the first character. With normal parsers this aspect is opaque.

Some API entry points in hotspots only accept such a kind of parser and not the base trait, when they are in a hotspot of the parsing logic. You normally do not have to worry about this, as most of the likely candidates for defining the start condition of an inline construct would satisfy the condition.

But when you get a compiler error `expected PrefixedParser[T], got Parser[T]` you know why. You tried to pass a non-optimizable parser where the library does not accept it. This prevents severe performance degradations just because an extension has been installed.

As a rule of thumb, all text parsers that do not A) build on negation (like `oneNot` or `someNot`) or B) allow for empty results like all `anyOf`, `anyNot` or `anyWhile` satisfy this condition.

This design follows the principle of enabling decent performance, but not at the cost of ease of extensibility, one key capability of the library.

Laika's HOCON API

Laika comes with its own HOCON parser and the corresponding API is used throughout the document model. It fully supports the **HOCON specification**, albeit based on its own parser implementation.

Why Laika Uses HOCON

There is a little irony in that Laika went the extra mile to introduce its own HOCON parser even though this project actually does not recommend the use of external files for application configuration!

Problems with Configuration Files

File-based application configuration was usually introduced with the promise of being able to relaunch the application with different configurations without rebuilding. However, most real-world deployment processes involve a git commit of any configuration changes and a subsequent rebuild on a CI server. This way this mechanism never delivers on its initial promise, but instead introduces several problems: the indirection of defining values in one place and accessing them in code and the stringly nature of the configuration which tends to be error-prone compared to a more type-safe approach.

For this reason newer configuration libraries like Ciris by-pass the file approach entirely. The motivation and some use cases are introduced in this **presentation**.

Likewise Laika encourages programmatic configuration approaches for all its global configuration options, and in fact none of the code examples in the manual for this type of configuration show any HOCON-based approaches.

Laika's Use Case for HOCON

So why is there are a custom HOCON parser in Laika then? The reason is that its primary use case in the library is not around global configuration. An input tree (usually obtained from one or more input directories in the file system) is a hierarchical structure, and every sub-directory or individual document can override some configuration aspect, as shown in **Where Laika Uses HOCON**. On top of that, Laika has an extension mechanism called directives (see **Implementing Directives** for details) and for its syntax which often allows to add attributes to a directive declaration HOCON is a natural fit.

On such a fine-grained level, programmatic configuration is not practical, and HOCON's format is an attractive solution thanks to its concise and flexible syntax and its support for substitution definitions where a declaration in a document or directive attribute section can directly refer to something defined globally or in a parent scope.

Why a New Parser?

After the question why HOCON has been chosen is answered, there is still a follow-up question about the need to re-implement an entirely new parser. One reason is that the original implementation (the **Typesafe Config** library) and the projects that wrap around it or derive from it do not meet Laika's requirement for

full referential transparency. They might throw exceptions or perform side effects without exposing it in the method signature. Another reason is that Laika extends the capability of what kind of data can be held in a configuration node. It allows an AST node to be assigned to a key in a `Config` instance so that it can be referred to in markup or templates. One of the key features, the way how a template merges the content of its associated markup document, is based on this extension, which allows the use of a substitution reference (`${cursor.currentDocument}`) for this task.

Finally, parsing is a central aspect of a lot of Laika's functionality anyway and it even comes with its own parser combinators. If you examine the HOCON parser implementation you will notice that it's a very lightweight and small module compared to the entire Laika code base.

Where Laika Uses HOCON

Multiple types in the document tree model have a `config` property holding the configuration for that scope:

- A `DocumentCursor` instance passed to directive implementations and custom rewrite rules. Probably the most likely instance you access a `config` property through.

- A `Document` instance, populated from the configuration header of the markup document

- A `TemplateDocument` instance, populated from the configuration header of the template document

- A `DocumentTree` instance, populated from the `directory.conf` file in the corresponding directory

- Each directive can optionally have a HOCON attribute block

Whenever the corresponding configuration header or file is missing, an empty `Config` instance will be used.

The `Config` instances will be populated with values supplied by the library in the `laika.*` and `cursor.*` namespaces. The former holds configuration values and the latter navigation info for the current document. An additional namespace `helium.*` will be used if you use the default theme, containing theme configuration and some pre-built AST nodes.

Any user-supplied values will be available, too, and should live in any namespace other than the two reserved ones.

Reading from Config Instances

Once you obtained access to a `Config` instance, either through one of the objects listed above or by Creating a Config Instance yourself, reading from it is quite straightforward. You have to provide the key you want to read and the type you expect:

```scala
val config: Config = ???

val version = config.get[String]("project.version")
```

The type conversion is based on a matching `ConfigDecoder[T]` which must be in implicit scope. It's a mechanism you might be familiar with from JSON libraries.

The returned value is of type `Either[ConfigError, T]`. It will be a `Left` when either the type conversion fails or the key is not defined.

Laika comes with decoders for basic types like `Boolean`, `String`, `Int`, `Path` or `Date` and sequences and maps of them. They are in the companion, therefore do not require any imports.

The `Path` decoder deals with the type representing Laika's virtual path. It is convenient in that it resolves relative paths to absolute paths based on the origin of the definition. See **Virtual Tree Abstraction** for details.

There are variants of the `get` method shown above: there is an overload that allows to pass a default value as the second argument that will be used when there is no value for the specified key. There is a `getOpt` variant that turns the result into `Either[ConfigError, Option[T]]` and there is a `hasKey` method to check for the existence of a key.

See **Config** for the full API documentation.

Creating Decoders and Encoders

Decoders are required for reading from a `Config` instance as shown in the previous section, the most common scenario. Encoders are needed when you populate a `Config` instance programmatically as shown in **Programmatic Building**.

ConfigDecoder

A decoder for a simple type is quite straightforward, it usually piggy-backs on an existing decoder. Let's assume you have a `Color` enum, with a constructor to create instances from a string:

```scala
object Color {
  def fromString (value: String): Option[Color] = ???
}
```

You can then flatMap on the string decoder to obtain a Color decoder:

```scala
implicit val colorDecoder: ConfigDecoder[Color] =
  ConfigDecoder.string.flatMap { str =>
    Color.fromString(str)
      .toRight(DecodingError(s"Unsupported color name: $str"))
  }
```

Now let's assume you need a decoder for a case class with the following shape:

```scala
case class Person (name: String, age: Int, city: Option[String])
```

For mapping a HOCON object to a Scala case class you would usually build on top of the `config` decoder, which decodes a nested object into an instance that has the same API for querying values as the root.

```scala
implicit val decoder: ConfigDecoder[Person] =
  ConfigDecoder.config.flatMap { config =>
    for {
      name <- config.get[String]("name")
      age  <- config.get[Int]("age")
      city <- config.getOpt[String]("city")
    } yield {
      Person(name, age, city)
    }
}
```

ConfigEncoder

We are going to demonstrate how to write encoders for the same objects we used for our decoder examples.

We can encode our `Color` enumeration, assuming it has a `name` property, like this:

```scala
implicit val colorEncoder: ConfigEncoder[Color]
 = ConfigEncoder.string.contramap(_.name)
```

For our Person case class we can use the convenient shortcuts in `ObjectBuilder`:

```scala
implicit val encoder: ConfigEncoder[Person] = ConfigEncoder[Person] { person =>
  ConfigEncoder.ObjectBuilder.empty
    .withValue("name", person.name)
    .withValue("age", person.age)
    .withValue("city", person.city)
    .build
}
```

The builder deals with optional values by omitting the property altogether when it's empty.

Automatic Derivation

Laika currently does not contain functionality for automatically deriving encoders or decoders for case classes. One reason is that in contrast to JSON libraries the likeliness you need to map larger structures is much smaller. Therefore the amount of boilerplate is usually tolerable.

Secondly we are so close to a Scala 3 release that will make automatic derivation much easier, that the step of building something on top of shapeless for Scala 2 (and paying the compile time tax) or writing a low-level Scala-2-style macro does seem very unattractive.

Automatic derivation will be supported once the Laika code base moves to Scala 3.

Creating a Config Instance

The most common use cases for `Config` instance in Laika are read access. But there may be scenarios where you want to create new instances yourself. One would be when you create an entire `DocumentTree` programmatically instead of parsing from sources. It is entirely possible in Laika to feed renderers for EPUB, PDF and HTML solely with content generated in-memory. The other use case would be if you want to use Laika's HOCON parser completely independently from its other features.

There are two ways to create a `Config` instance. One is based on parsing HOCON string input with a `ConfigParser`, the other is to assemble configuration values programmatically with a `ConfigBuilder`. The former mostly exists for scenarios where you need to work with files, whereas the latter should be preferred for creating configuration data in-memory.

Programmatic Building

A `ConfigBuilder` allows to assemble arbitrary values as long as they have a `ConfigEncoder` in scope.

```
val config = ConfigBuilder.empty
  .withValue("laika.epub.coverImage", "/images/epub-cover.jpg")
  .withValue("laika.pdf.coverImage", "/images/pdf-cover.jpg")
  .build
```

The first parameter is the key to assign to the value, the second is the actual value which will be converted based on an implicit `ConfigEncoder` in scope.

Laika comes with encoders for basic types like `Boolean`, `String`, `Int`, `Path` or `Date` and sequences and maps of them. They are in the companion, therefore do not require any imports.

You can alternatively create your own encoder as shown above.

If you have a fallback instance, you can pass it to the constructor:

```
val parentConfig: Config = ???

val config = ConfigBuilder.withFallback(parentConfig)
  .withValue("laika.epub.coverImage", "/images/epub-cover.jpg")
  .withValue("laika.pdf.coverImage", "/images/pdf-cover.jpg")
  .build
```

The fallback will be used for resolving any values not present in the current instance.

Finally, if you are building a `Config` instance that you want to assign to a `Document` instance in cases where you build an entire tree programmatically, you also have to provide a correct `Origin` instance:

```
val doc: Document = ???
val docOrigin: Origin = Origin(Origin.DocumentScope, doc.path)

val config = ConfigBuilder.withOrigin(docOrigin)
  .withValue("laika.epub.coverImage", "/images/epub-cover.jpg")
  .withValue("laika.pdf.coverImage", "/images/pdf-cover.jpg")
  .build

val finalDoc = doc.copy(config = config)
```

This is essential for resolving relative paths defined in that configuration correctly.

Parsing HOCON

The `ConfigParser` has a very simple API:

```
val hoconInput: String = ???

val config: Config = ConfigParser
  .parse(hoconInput)
  .resolve()
```

The `parse` step creates an interim model of unresolved configuration values. This is necessary because the HOCON format supports substitution references and the corresponding values do not need to originate from the same instance. The `resolve` step then finally creates a `Config` instance, resolving and validating all references.

If you have a fallback instance, you can pass it via `resolve`:

```
val hoconInput: String = ???
val parentConfig: Config = ???

val config: Config = ConfigParser
  .parse(hoconInput)
  .resolve(fallback = parentConfig)
```

The fallback will be used for resolving any values not present in the current instance.

Finally, if you are building a `Config` instance that you want to assign to a `Document` instance in cases where you build an entire tree programmatically, you also have to provide a correct `Origin` instance:

```scala
val hoconInput: String = ???
val doc: Document = ???
val docOrigin: Origin = Origin(Origin.DocumentScope, doc.path)

val config: Config = ConfigParser
  .parse(hoconInput)
  .resolve(origin = docOrigin)

val finalDoc = doc.copy(config = config)
```

This is essential for resolving relative paths defined in that configuration correctly.

There is currently no API for conveniently reading HOCON from files. You would need to do the file IO yourself before feeding the parser. Should the HOCON parser become a standalone micro-lib, it would definitely get a `ConfigLoader`. But within Laika's own usage all IO is performed by the logic in the `laika-io` module that also deals with loading of markup files and templates.

Standard Directives

Laika comes with a small number of built-in directives intended to cover the most common scenarios.

Some of them can only be used in templates, whereas others can also be used in text markup, as stated for each individual directive below.

For details on creating your own directives, see **Implementing Directives**.

Navigation

This set of directives helps with building complex navigation structures or serve as shortcuts for creating links.

The directives in this section are all documented in detail in the **Navigation** chapter and only listed here for the sake of completeness.

`@:navigationTree`

Can be used in templates and as block elements in text markup.

This flexible directive allows to freely combine auto-generated navigation trees based on your input tree's structure with manual entries.

It is documented in detail in **Generating Navigation Trees** in the Navigation chapter.

`@:breadcrumb`

Can be used in templates and as block elements in text markup.

This standard breadcrumb component generates a simple, flat list of links reflecting the directory structure from the root to the current document. See **Breadcrumbs** in the Navigation chapter for details.

`@:api`

Can be used as inline element and within links in text markup.

Serves as a shortcut for creating links to API documentation, e.g. `@:api(laika.api.Transformer)`.

See **Linking to API Documentation** for details.

`@:source`

Can be used as inline element and within links in text markup.

Serves as a shortcut for creating links to the source code of types, e.g. `@:source(laika.api.Transformer)`.

See **Linking to Source Code** for details.

Applying Styles

Laika already produces a small set of class attributes in the rendered output, e.g. `nav` and `level1` for a navigation element on the first level. But you can alternatively assign style hints to markup explicitly.

`@:style`

Can be used in block or span elements in text markup.

Adds a style to the root element of the body of the directive, that can be used by renderers to change the appearance of the rendered element.

For HTML and EPUB output it will get rendered as a `class` attribute that can be used in CSS declarations. For PDF it can also be used in Laika's **CSS for PDF** support, even though there is no interim HTML output.

Block directive:

```
@:style(subtitle)

This paragraph gets the
subtitle style applied.

@:@

While this paragraph does not.
```

Span directive:

```
Burrito - @:style(price) £3.50 @:@.
```

Markup Blocks

The directives in this section can only be used in place of block-level elements in text markup.

`@:image`

Can be used in block or span elements in text markup.

The image directive is an alternative to the native syntax for including images that supports additional attributes.

```
@:image(logo.png) {
  intrinsicWidth = 64
  intrinsicHeight = 64
  style = logo
  alt = Project Logo
  title = Tooltip Text
}
```

Note that the path, like everything in Laika, is interpreted to be within the virtual path of the input tree you configured. It can be relative or absolute, in the latter case it would start with a / and See **Virtual Tree Abstraction** for details.

All other attributes shown in the example are optional:

- `intrinsicWidth` and `intrinsicHeight` can be used to avoid "layout shift" by telling the browser the size of the image before it is loaded. As the attribute names imply, they are not intended to be used for the actual display size. For this purpose you should use the `style` attribute and provide corresponding CSS for sizing the image.

- `alt` and `title` are pass-through attributes for the generated image tag. They are both ignored for PDF output.

@:callout

Can only be used in block elements in text markup.

A callout is a decorated block element that stands out from the surrounding paragraphs.

In markup the directive simply surrounds the content of the callout:

```
@:callout(warning)

This text appears inside the decorated callout box.
Any block level elements can be used as the content,
including lists, tables or code blocks.

@:@
```

When using the Helium theme it will be rendered as shown in the example below.

> ⚠
>
> This text appears inside the decorated callout box. Any block level elements can be used as the content, including lists, tables or code blocks.

Helium comes with pre-defined styles and icons for the attributes `info`, `warning` and `error`.

`@:select`

The select directive is a special directive that allows to create alternative versions of the same documentation, for example one with Scala code examples and one with Java. Or as in the case of Laika's own documentation, one showing configuration setup with sbt and the other through the library API.

When using the default Helium theme these alternatives will be rendered as tabs in the website:
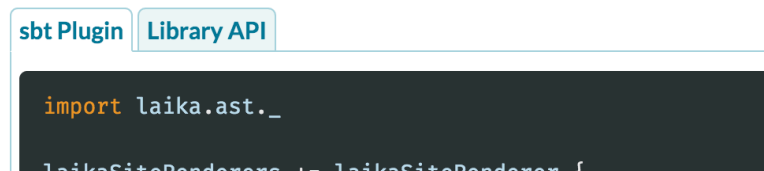
While for EPUB and PDF output it triggers the generation of separate books for each of the alternatives and offers them on the download page:
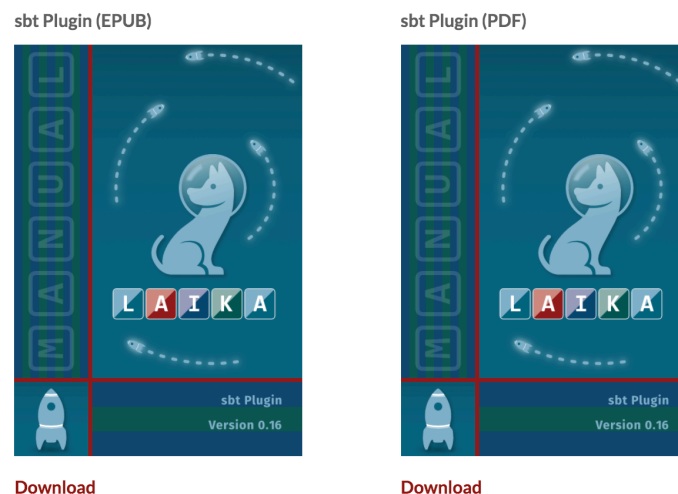
This accounts for the fact that tabs would not make much sense when printed on paper!

Before using this directive you need to put a bit of configuration in place. Laika expects the labels to use for the alternatives to be declared centrally, so that they do not need to be repeated with each use of the directive.

This is how the settings for Laika's manual look:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue(Selections(
    SelectionConfig("config",
      ChoiceConfig("sbt", "sbt Plugin"),
      ChoiceConfig("library", "Library API")
    ).withSeparateEbooks
  ))
```

The configuration above uses three identifiers: `config` as the name of the group of alternatives, and `sbt` and `library` as the name of the two choices together with their labels.

With this configuration in place the directive can be used like this:

```
@:select(config)

@:choice(sbt)
```scala
laikaConfig := LaikaConfig.defaults.strict
```

@:choice(library)
```scala
val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)
  .strict
  .build
```

@:@
```

The `@:choice` directives are child directives called "separator directives" that can only be used inside the corresponding parent directive to mark specific sections within the directive's body.

@:fragment

Can only be used in block elements in text markup.

Produces a block element which is not part of the main body of the markup document. Instead it can be referred to by `${cursor.currentDocument.fragments.<fragmentName>}`.

This allows to keep some sections of your document separate, to be rendered in different locations of the output, like headers, footers or sidebars.

This directive is explained in detail in the **Document Fragments** chapter.

@:format

Can only be used in block elements in text markup.

The body of such a directive will only be included into the output by renderers for the specified type:

```
@:format(html)

This text only appears in the HTML output,
not in PDF, EPUB or any other format.

@:@
```

HTML Templates

These directives can only be used in HTML or EPUB templates, as they drive the inclusion of tags in the HTML `<head>` section.

If you use the Helium theme and do not design your own templates or themes, you do not need to use these directives directly, as the Helium template already contain them. In that case you can control which CSS and JS files to link via the theme's configuration as described in **Auto-Linking CSS & JS Files**.

@:linkCSS

Causes the automatic inclusion of `<link>` tags for CSS files found in the input tree.

```
@:linkCSS
```

When used without attributes as in the example above it will simply link any CSS file found in the input tree.

If you only want to pick files from specific locations, you can use the `paths` attribute:

```
@:linkCSS {
  paths = [ /shared, /css ]
}
```

The order of the link tags will reflect the order of paths specified. You can point to individual files or directories. In the latter case all files found in that directory or any sub-directory will be included.

Note that the paths, like everything in Laika, are within the virtual path of the input tree you configured. See **Virtual Tree Abstraction** for details.

@:linkJS

Causes the automatic inclusion of `<script>` tags for JavaScript files found in the input tree.

```
@:linkJS
```

When used without attributes as in the example above it will simply link any JavaScript file found in the input tree.

If you only want to pick files from specific locations, you can use the `paths` attribute:

```
@:linkJS {
  paths = [ /shared, /js ]
}
```

The order of the script tags will reflect the order of paths specified. You can point to individual files or directories. In the latter case all files found in that directory or any sub-directory will be included.

Note that the paths, like everything in Laika, are within the virtual path of the input tree you configured. See **Virtual Tree Abstraction** for details.

Conditionals and Loops

These directives are kept very simple on purpose. For example they do not include any expression language right now that would allow you to express conditions like `@:if(catalogue.items > 5)`, but instead only check for booleans or non-empty values. For more complex use cases it's often easier to implement your own directive instead of adding a lot of logic to your templates.

`@:for`

Can only be used inside templates.

Executes the body of the directive multiple times for each element of a non-empty array-type entry in the configuration. It does not include the body at all for the values `null`, `false`, `None`, empty strings and empty collections, and exactly once for all other values.

In case of non-empty values their properties will be made available for use inside the body part without any prefix:

```
@:for(catalog.products)

<li><a href="#${_.id}">${_.description}</a></li>

@:@
```

In this example `id` and `title` are properties of the `catalog.products` property that the user defined in the configuration. See **User-Defined Variables** for details on defining them.

You can also specify a fallback, a body part that gets executed for empty values:

```
@:for(catalog.products)
<li><a href="#${_.id}">${_.description}</a></li>

@:empty
<p>No products available.</p>

@:@
```

`@:if`

Can only be used inside templates.

Executes the body of the directive exactly once for a Boolean `true` or the strings `true`, `on`, `yes` and `enabled`.

```
@:if(showSidebar)
<div class="sidebar">...</div>
@:@
```

In this example `showSidebar` is a custom configuration entry that the user defined in the configuration. See **User-Defined Variables** for details on defining them.

You can also specify a fallback with `@:else`, or even secondary conditions with `@:elseIf`:

```
@:if(showSidebar)
<div class="sidebar">...</div>

@:elseIf(showInfobox)
<div class="infobox">...</div>

@:else
<p>Something else</p>

@:@
```

PDF Output

The following directive is only processed with PDF output and ignored with other formats.

`@:pageBreak`

Can only be used in block elements in text markup.

The directive does not have any attributes or body elements and simply inserts a page break into the final PDF document.

```
@:pageBreak
```

Comments

Since Markdown does not have any native comment syntax, Laika offers a directive to add todo's to text markup:

`@:todo`

Can be used in block and span elements in text markup.

```
@:todo(add diagram)
```

The directive will be ignored by all renderers and won't show up in any output documents.

Substitution Variables

Substitution Variables can used in text markup documents and in templates for any output format.

Syntax

The syntax is part of Laika's HOCON support and matches the **HOCON spec**.

A mandatory reference can be inserted with the key of the variable enclosed between $\{ and \}:

```
${cursor.currentDocument.title}
```

If such a value is undefined, the transformation will fail (unless you tweaked the configuration for **Error Handling**).

An optional reference can be expressed by adding an additional ? right before the variable key:

```
${?cursor.currentDocument.title}
```

If an optional reference points to an undefined value, the reference in the template is substituted with an empty string.

Pre-Defined Values

Laika provides a set of pre-defined variables in three namespaces. One is the `cursor` namespace that holds information about the current document and its siblings and parents. The second is `laika` holding all values coming from Laika's internal configuration. Finally the `helium` namespace which is only populated if you use Laika's default theme called Helium, contains the theme configuration and some pre-built AST nodes.

Cursor Values

Values from the cursor namespace expose information about the current document and its siblings and parents. They can be used in templates or markup documents to create links or insert content. If you need to create complex navigation structures though, it is usually best to prefer **Implementing Directives** over complex template logic.

Internal processing in Laika is not stringly, but based on traversing and modifying ASTs. Therefore, if a substitution variable listed below is described as holding an AST, the node in the AST representing the variable will be replaced by this referenced AST. Rendering of the AST happens later as a final step in the processing pipeline.

This is a complete list of values exposed in the `cursor` namespace:

- `cursor.currentDocument`: A pointer to the currently processed document. In templates this is populated for each document the template is applied to - it refers to the markup document, not the template document itself. Sub-keys are:

  - `content`: The entire AST of the markup document, see **Inserting Content from Markup Documents** for details.

  - `fragements`: The AST of all named fragments declared in the markup document, under their respective keys, e.g. `cursor.currentDocument.fragments.sidebar` for a fragment declared with `@:fragment(sidebar)` inside the markup document. See **Document Fragments** for details.

  - `title`: the AST of the title of this document - including formatting.

  - `path`: the absolute (virtual) path of the document inside the input tree.

- Access to surrounding documents via `cursor.parentDocument`, `cursor.previousDocument`, `cursor.nextDocument`, `cursor.flattenedSiblings.previousDocument` and `cursor.flattenedSiblings.nextDocument`. Any of these references may be empty, depending on the position of the current document.

  The `flattenedSiblings` differ from the other pointers in that they go beyond the current chapter. If the current document is the last in the current chapter (directory), then `cursor.nextDocument` will be empty, but `cursor.flattenedSiblings.nextDocument` will point to the first document in the next chapter. The `flattenedSiblings` are useful if you want to create a linearized, book-like navigation.

  Sub-keys of these document pointers are:

  - `title`: the AST of the title of the document - including formatting.

  - `absolutePath`: the absolute path in the virtual input tree as a string, e.g. `/herbs/parsley.md`.

  - `relativePath`: the path relative to this document as a string, e.g. `../parsley.md`.

- `root.title`: The title of the root node, usually the title of the website or e-book.

Laika Configuration Values

These are usually not accessed in user templates and mostly intended for Laika's own internal processing, but are nevertheless exposed like any other value in the configuration.

- `laika.<format>.metadata`: holds the **Metadata** specified in configuration.

- `laika.links`: holds navigation configuration for **Global Link Definitions**, **Linking to API Documentation** and **Disabling Validation**.

- `laika.autonumbering`: the configuration for auto-numbering documents and sections. See **Auto-Numbering** for details.

- `laika.pdf` and `laika.epub`: e-book configuration, see **E-Books (EPUB & PDF)** for details

- `laika.titleDocuments`: configures the names of documents that should be treated as title documents. See **Title Documents** for details.

User-Defined Values

In addition to the library's pre-defined values users can define their own values in any scope.

The only reserved namespaces for variable keys are `cursor` (e.g. `cursor.currentDocument.title`), `laika` (e.g. `laika.pdf.coverImage`) and `helium` for the default theme. You can freely choose any key outside of these three namespaces.

Programmatic Definition

You can add arbitrary configuration values when building a `Parser`, `Renderer` or `Transformer`:

```
laikaConfig := LaikaConfig.defaults
  .withConfigValue("project.version", "2.4.6")
```

You can refer to these values in templates and text markup:

```
The latest stable release is ${project.version}.
```

Values declared in a builder instance are available globally, but can be overridden by HOCON definitions in narrower scopes, as shown in the next section.

HOCON Definition

Programmatic configuration is recommended over file-based HOCON configuration wherever possible, but if you need to associate values with a particular directory, text markup document or template, instead of declaring them globally, you can define them in the corresponding locations:

- a configuration file with the name `directory.conf` in any input directory

- a configuration header in a template

- a configuration header in a text markup document

Just be aware that these values are then not available globally, but only in the corresponding scope they were defined in.

Example for defining a variable in a configuration header in a text markup document:

```
{%
  project.version = 2.0.2
%}

Document markup here...
```

This value can then be accessed within that document or inside a template applied to that document with the usual syntax:

```
The latest stable release is ${project.version}.
```

Programmatic Access

The same variables that you can access with HOCON-style substitution references in templates and markup files, are also available programmatically if you are working with a document tree. The `DocumentCursor`, `DocumentTree`, `Document` and `TemplateDocument` types all have a `config` property that exposes those values:

```
val doc: Document = ???
val version: Either[ConfigError, String] =
  doc.config.get[String]("project.version")
```

See **Config** for details about the Config API.

Spec Compliance

This chapter gives a brief overview on how closely Laika adheres to the various formats it parses.

Markdown

Laika sticks to the original **syntax description** and incorporates the **test suite** from the PHP Markdown project which is a slightly expanded version of the original suite from John Gruber.

In cases where both the spec and the test suite are silent on how to handle certain edge cases, the **Babelmark** tool has been consulted and usually the approach the majority of available parsers have chosen has been picked for Laika, too.

Test Suite

The testing approach in Laika is adapted to cater for the library's design goals, one of which is decoupling input and output formats. Most existing, official test suites are providing inputs in text markup and output in HTML. In cases where the output differs from Laika's built-in HTML renderer, the renderer is adjusted with overrides just for the test. This is acceptable because these subtle differences do not represent a semantic difference. The only way to avoid these kind of "little cheats" would be to have a separate HTML renderer for each supported text markup format. But this would be undesirable, in particular in cases where users mix documents with different markup formats in the same input directory.

Verbatim HTML

One major difference to standard Markdown is that the parsing of verbatim HTML elements is not enabled by default, as Laika discourages the coupling of input and output formats, but it can be switched on if required.

See **Raw Content** for examples on how to enable verbatim HTML in the sbt plugin or the library API.

When this support is switched on, it follows the original spec, including the support for text markup and HTML syntax being interspersed in the input.

GitHub Flavored Markdown

Laika supports the syntax of GitHubFlavored Markdown through an `ExtensionBundle` that must be enabled explicitly. These are the parsers this extension adds to standard Markdown:

- strikethrough (**spec**)

- auto-links (urls and email addresses - **spec**)

- fenced code blocks (**spec**)

- tables (**spec**)

Subtle Differences to the GitHub Specification

- **Spec Alignment**: The Laika implementation is an extension of classic, standard Markdown, in a similar fashion as GitHub Flavored Markdown had initially been defined. However, GitHub's spec has since moved on and is now based on the **CommonMark** spec. This should not make a huge difference for the most common use cases as CommonMark stays pretty close to classic Markdown and the syntax that has since moved to CommonMark (e.g. fenced code blocks) is included in Laika's extension. You'll probably only notice differences around any of the subtle lower-level ambiguities in Markdown's syntax.

- **Auto-Links**: The parsing of URIs in auto-links is based on the relevant RFCs and not on the rather informal description in the GitHub spec. This should not make any difference for the most common use cases. The RFC based URI parser has been part of Laika for years (as reStructuredText natively supports auto-links) and its higher level of correctness justifies the bypassing of the informal description of GitHubs spec.

- **Fenced Code Blocks** need a preceding blank line to be recognized by Laika's parser for now. This is due to temporary technical limitations which will be lifted before the 1.0 release. It will require additional hooks for how extension parsers register with the host language to allow to specify a line test that causes interruption of a paragraph.

- **Tables**: Since Laika is a tool that uses an internal AST that abstracts away the features of a specific output format, it does not follow the exact syntax for HTML output as shown in the GitHub spec. Specifically it does not render table cells using the deprecated `align` attributes. Instead it renders the cells with classes (`alignLeft`, `alignCenter`, `alignRight` or none) so that the cells can get styled in CSS.

CommonMark

Laika does not yet integrate the official CommonMark test suite. This step is planned for a release shortly after the final 1.0 version is reached.

In practice the differences should be minor as CommonMark is a specification that builds on top of the original Markdown spec plus some aspects of GitHub Flavor which Laika both supports. It mostly removes some ambiguity and adds detail to some of the under-specified features of classic Markdown.

reStructuredText

The reStructuredText project is part of Python's Docutils project. It is also more strictly defined than Markdown, with a detailed **specification** and clearly defined markup recognition rules.

Apparently there is no official test suite for reStructuredText, therefore to add a realistic test to the Laika test suite a full transformation of the reStructuredText specification itself is integrated into Laika's test suite.

Supported Standard Directives

Directives are an extension mechanism of reStructuredText and the reference implementation supports a set of standard **standard directives** out of the box.

Out of this set Laika supports the following:

- Admonitions: `attention`, `caution`, `danger`, `error`, `hint`, `important`, `note`, `tip`, `warning` and the generic `admonition`

- Document Structure: `contents`, `sectnum`, `header`, `footer`, `title`, `include`

- Containers: `compound`, `container`, `topic`, `sidebar`, `rubric`, `table`, `parsed-literal`, `code`

- Quotations: `epigraph`, `highlights` and `pull-quote`

- Images: `figure` and `image`

- For Substitution Definitions: `replace`, `date`, `unicode`

- Other: `meta`, `raw`

The following limitations apply to these directives:

- `code` does currently not support syntax highlighting (it allows to set the language though, so client-side highlighters can be integrated if required).

- `sectnum` does currently not support the `prefix`, `suffix` and `start` options.

- `include` does not support any of the options apart from the filename, due to the way document trees are traversed very differently in Laika.

- `raw` does not support the `file` or `url` options.

- `unicode`: does not support the various trim options, as that would require modifying adjacent elements (and no other directive has this requirement, therefore API/implementation changes did not seem justified)

- `date`: Uses the patterns of `java.text.SimpleDateFormat` instead of Python's `time.strftime` function.

Supported Standard Text Roles

Text roles are a second extension mechanism for applying functionality to spans of text and the reference implementation supports a set of standard **standard text roles** out of the box.

Out of this set Laika supports the following:

- `emphasis`

- `strong`

- `literal`

- `subscript` (and `sub` alias)

- `superscript` (and `sup` alias)

- `title-reference` (and `title` alias) - the default role

- `raw` (+ format option)

Unsupported Extensions

The following extensions are not supported:

- `math`, `csv-table`, `list-table`, `target-notes` and `class` directives

- `math`, `pep-reference` and `rfc-reference` text roles

There are various reasons for excluding these extensions, some of them being rather technical. For example, the `target-notes` and `class` directives would require processing beyond the directive itself, therefore would require new API. Others, like the `pep-reference` text role, seemed too exotic to warrant inclusion in Laika.

Raw Content Extensions

Two of the supported standard extensions, the `raw` directive and the `raw` text role, embed content in the target format in text markup. Like with verbatim HTML for Markdown, these extensions are disabled by default, as Laika discourages the coupling of input and output formats, but it can be switched on if required.

See **Raw Content** for examples on how to enable these extensions in the sbt plugin or the library API.

Implementing a Custom Directive

Laika comes with a concise and typesafe DSL to declare custom directives and text roles. It is fully documented in the scaladoc for **Directives** and **TextRoles**.

HOCON

Laika fully supports the official **HOCON specification**, based on its own parsers (it does not wrap the Typesafe Config library).

HOCON in Laika is supported in various places:

- Directory configuration files

- Configuration headers in text markup documents and templates

- Attribute sections in directive declarations

- Substitution references compliant with the HOCON spec are supported anywhere in text markup and templates, including the syntax for optional references, e.g. `${?path}`.

For more information on how HOCON is used in the library see **Laika's HOCON API**.

The only fairly minor exception in spec compliance is the deliberate decision not to support circular references. Direct self references are supported (e.g. `path = ${path} [img, css]`) as they cover a very common use case, but **a** referring to **b** referring to an earlier definition of **a** is not. The cost of additional complexity in the implementation this would require seems disproportional to the little use supporting this edge case would provide. Convoluted circular references are also harder to read and error-prone when definitions move.

If you think this should be supported, please open a ticket and explain your reasoning and your use case.

Release Notes

0.16.1 (Sep 12, 2020)

- New Lightweight Theme called Helium

  - Ready-to-use styles for the 3 major output formats: HTML, EPUB, PDF

  - No dependency on 3rd-party CSS or JavaScript frameworks, just a minimal amount of hand-crafted CSS and JS.

  - Responsive design of the site output.

  - Define font resources, color sets, layout and other details with the Scala configuration API for Helium

  - Support font embedding for EPUB and PDF out of the box

  - Supports auto-linking of all CSS and JavaScript files from your input tree in the HTML output for websites and EPUB, search paths can be restricted if necessary.

  - Obtain more low-level control over the output by overriding the theme's CSS.

  - Includes an auto-generated main navigation bar and a page navigation with configurable depth.

  - Anchors on mouse-over for headlines, providing the URL to the section.

  - Favicon support.

  - Support for font icons as well as a set of default icons included in the theme.

  - Integrated download page offering the site content as EPUB and PDF.

  - Website landing page tailored for software documentation sites.

- New API for Theme Authors

  - The Theme API allows to create 3rd-party themes by pre-populating the input tree with styles, scripts, images, font resources as well as providing extensions for the Laika Core features in the form of the existing `ExtensionBundle` API.

- New API for freely composing inputs

  - Individual files, in-memory strings or streams can now be freely combined and "mounted" to a logical path within Laika's input tree.

  - Add support for classpath resources.

  - Add support for providing inputs as a pre-built document AST, bypassing the parsing step altogether.

  - Supported for user code (when specifying the inputs to transform) as well as for theme authors.

- New Directives

  - `@:callout` produces decorated text blocks with background color and icon.

  - `@:image` enhances the options native markup provides by allowing to specify the intrinsic width and height of images to avoid layout shift in browsers as well as assigning a style for controlling the display size via CSS.

  - `@:select` is a powerful directive to create alternative versions of your content, e.g. for Scala vs. Java APIs or for sbt vs Mill build examples. In the rendered website these choices are available via tabs, for EPUB and PDF they will lead to separate artifacts containing only one of the choices.

  - `@:source` is a link directive, allowing to specify a fully qualified classname, similar to the existing `@:api` directive, but linking to the source (e.g. on GitHub) instead of the API documentation.

  - `@:linkCSS` and `@:linkJS` can be used in HTML template files in the `head` section for auto-linking all CSS and JS files found in the input tree (or restricted to specific directories only).

  - `@:todo` is a little helper directive to overcome the problem that Markdown does not have comment syntax. Renderers will simply ignore the content of the directive.

- EPUB Support

  - Tweak defaults to accommodate for the fact that some popular readers like iBooks do not support the full standard for their navigation menus. Default navigation depth is now only 2 for EPUB output.

  - Introduction of special suffixes to distinguish CSS for EPUB from site CSS: `*.epub.css` will only be used for EPUB, `*.shared.css` will be used for EPUB and site, while all other CSS files will only be used for the site.

- API Change for Theme Support

  - All transformers, parsers and renderers from the `laika-io` module are now provided as a cats-effect `Resource`. This change was necessary as themes are themselves passed to transformers as a `Resource` as they might require side-effecting initialization logic. See the **Migration Guide** for the (trivial) changes that are necessary for code building transformers.

- Bugfixes

  - EPUB: the library could produce invalid XML metadata files for the EPUB container under some circumstances, e.g. a headline staring with a number or containing an ampersand.

  - The `excludeFromValidation` flag was ignored in some scenarios.

  - An AST rewrite rule returning an explicit `Retain` could override a `Replace` action from a previous rule.

- Project Maintenance

- Update dependencies to cats 2.2.0, cats-effect 2.2.0 and ScalaTest 3.2.2.

- Solely use sbt's slash syntax in Laika's build and all configuration examples for Laika's sbt plugin

- Remove all deprecated classes and methods.

- Remove legacy directive syntax that had been deprecated since 0.12.

- This release is identical with 0.16.0 which had a broken `laika-pdf` artifact.

0.15.0 (May 29, 2020)

- Enhanced Navigation Features

  - Internal references are now fully validated and dead links cause the transformation to fail by default.

  - Introduce global link definitions that allow to avoid repetition with centralized declarations of internal or external link targets.

  - Introduce new `@:navigationTree` directive that allows to combine auto-generated navigation trees with explicit manual entries. Deprecates the old `@:toc` directive.

  - Add new `@:breadcrumb` directive that produces a list of links from the root to the current document.

  - Introduce pluggable slug builder in `ExtensionBundle` to allow customizations of how a section headline is translated to an id.

  - Introduce "Link Directives", a new directive type that allows to define shortcuts for commonly used URL patterns, (e.g. `@:rfc(2022)`).

  - Add a new `@:api` link directive as a shortcut for linking to API documentation.

  - Change default names for title documents to `README.<suffix>` for the input for better GitHub integration and `index.<suffix>` for the output for better site navigation. Defaults can now be overridden in configuration.

  - Add methods for navigating siblings in Cursor API for document trees.

  - Laika now always picks the first header as a title, even when subsequent headers are on the same level.

  - Improved precision in detecting ambiguous references.

  - Internal overhaul/rewrite of the link resolver algorithm which was still the original design from 2013 and degraded in clarity over several feature additions.

- Configuration

- Introduce programmatic ways to declare all global configuration for both, sbt plugin and library API, which now allows to completely avoid configuration files unless per-directory configuration is required.

- Change HOCON configuration keys to always be in a namespace (e.g. `laika.title` or `cursor.currentDocument`), to reduce the likeliness of name clashes with user-defined variables.

- Error Handling

  - The presence of invalid nodes in the document AST before rendering now always triggers an abortion of the transformation with all messages from invalid nodes listed in the error result.

  - The new default behaviour can be overridden in configuration to switch to "visual debugging" where invalid nodes are rendered in their original position in the result.

- Changes in the Document AST related to navigation enhancements

  - Merge the functionality of `CrossLink` into `InternalLink`

  - Merge `ExternalLink` and `InternalLink` to `SpanLink`

  - Split `Temporary` marker trait into `Unresolved` and `Hidden`

- Manual

  - Complete rewrite of the manual which still had its initial structure from 2012.

  - Better separation of tutorial-style from reference-style sections.

  - New chapters about Laika's parser combinators, HOCON API, design goals, configuration options, navigation features, writing parser extensions and adding new syntax highlighters.

- Bugfix

  - The combination of the `laikaSite` task with the `laikaIncludeEPUB` setting led to a zero-byte result.

0.14.0 (Feb 28, 2020)

- Introduce support for Scala.js for the entire laika-core module:

  - Brings the complete functionality of Laika to Scala.js with the only exceptions being File/Stream IO and support for PDF and EPUB output

  - Eliminate the last, semi-hidden usages of Java reflection to enable Scala.js support

  - Avoid use of `java.time.Instant` in the shared base to not force another heavy dependency on Scala.js users

- Laika's integrated syntax highlighting:

  - Add support for Dotty, JSX/TSX, SQL, EBNF, Laika's own AST format

- Parser APIs:

  - Introduce `PrefixedParser` trait to make span parser optimizations implicit for 90% of the use cases and no longer require the developer to explicitly supply a set of possible start characters for the span

  - Introduce new parser for delimiters that generalizes typical checks for preceding and following characters in markup parsers (e.g. `delimiter("**").prevNot(whitespace)`), to reduce boilerplate for inline parsers

  - Introduce shortcuts for very common usages of text parsers (e.g. `oneOf('a','b')` instead of `anyOf('a','b').take(1)`)

  - Expand the API of the `Parser` base trait: add `source` method for obtaining the consumed part of the input instead of the result of the parser, and `count` to obtain the number of consumed characters

  - Introduce `laika.parse.implicits._` for extension methods for common parsers, e.g. `.concat`, `mapN`.

  - Introduce shortcut for repeating a parser with a separator

  - Deprecate all APIs that relied on implicit conversions

  - Deprecate some of the rather cryptic symbol methods on `Parser` in favor of named methods

  - `DelimitedText` does no longer have a type parameter

- AST changes and improvements:

  - Introduce RelativePath as a separate type in addition to the existing Path type which is now only used for absolute paths

  - Introduce a range of shortcut constructors for Laika's AST nodes, that allow you to write `Paragraph("some text")` instead of `Paragraph(Seq(Text("some text")))` for example

- Directives:

  - Cleaner syntax for default attributes which are no longer part of the HOCON attribute block for named attributes

- Demo App:

  - Cleanup from six to two (bigger) panels for improved usability

  - Use Laika's own syntax highlighting in the output

  - Allow to switch from JVM execution to Scala.js

  - Move from Akka HTTP to http4s

  - Integrate source into Laika's main repository

- Bugfix:

- Syntax highlighting for interpolated strings in Scala was broken in 0.13.0

- Fix a flaky test that relied on the ordering of HashMap entries

0.13.0 (Jan 26, 2020)

- Introduce integrated syntax highlighting based on the libraries own parsers

  - Resulting AST nodes for code spans are part of the document AST and can be processed or transformed like all other nodes

  - Works with any renderer, including PDF

  - Initially supported are Scala, Java, Python, JavaScript, TypeScript, HTML, CSS, XML, JSON, HOCON

  - Convenient base parsers for common syntax like string and number literals or identifiers to facilitate development of new syntax highlighters

- HOCON parser: add support for `include` statements, this final feature addition makes Laika's HOCON support fully spec-compliant

- New transformation hooks in the `laika-io` module for parallel transformers: `mapDocuments(Document => Document)`, `evalMapDocuments(Document => F[Document])` and the corresponding `mapTree` and `evalMapTree`

- Transformer introspection: introduce `describe` method for parsers, renderers and transformers and `laikaDescribe` setting in the sbt plugin that provides formatted information about the transformer setup and installed extensions

- sbt plugin: improved accuracy for caching logic for EPUB and PDF output that still works when the artifact name or version changes

- Upgrade dependency on cats-core to 2.1.0

0.12.1 (Dec 1, 2019)

- Fixes and Improvements for the new HOCON parser introduced in version 0.12.0

  - Significant improvements for error messages in HOCON parser

  - Fixes for nested self references, missing optional self references, and objects without '=' or ':' separator

- Parser Combinators: The '|' alternative parser now keeps the failure with the most processed characters and not the last one, for improved error messages

- Fix for script tag with attributes not being recognized in verbatim HTML in Markdown

0.12.0 (Oct 30, 2019)

- New laika-io Module

  - Functionality extracted from existing laika-core module

  - Contains File/Stream IO, EPUB output, Parallel transformations

  - Based on cats-effect type classes (in "Bring-Your-Own-Effect" style)

  - Leaves laika-core pure, preparing it for later support for Scala.js

- Referential Transparency

  - No method in the public API throws Exceptions anymore

  - The result of pure operations is provided by instances of `Either`

  - The result of side-effecting operations is provided by a return type of `F[A]`, with `F[_]` being an effect type from cats-effect

  - Eliminated all uses of runtime reflection

- Changes to the APIs for creating and running parsers, renderers and transformers

  - Necessary due to the changes listed above

  - See the migration guide for details and examples

- Changes to the Directive Syntax in Templating

  - The syntax of separators for the attribute and body sections have changed

  - HOCON syntax is now used for attributes

  - The old syntax is still supported, but will be removed at some point before the 1.0 release

- Changes to the Directive DSL for creating directives

  - `attribute(Default)` is now `defaultAttribute`

  - `body` is now either `parsedBody` or `rawBody`

  - Type conversions happen with the new `as` method: `attribute("title").as[String]`, based on the `ConfigDecoder` type class that is also used for the new Config API

  - Named body parts have been replaced by the more flexible Separator Directives

  - The built-in helper for mapping directive parts with different arity has been replaced by cats' `mapN`

- Refactoring of AST Rewrite API to be fully type-safe and avoid runtime reflection and exceptions.

- Return types are now more explicit (e.g. `Replace(newSpan)` instead of `Some(newSpan)`)

  - Rules for rewriting spans and blocks get registered separately for increased type-safety, as it is invalid to replace a span with a block element.

- Refactoring of the Render API to be referentially transparent

  - Also applies to the API for registering custom renderers for individual AST nodes

- New Config API and built-in HOCON parser

  - Removed the dependency on the Typesafe Config library and its impure Java API

  - Added a new lightweight and pure HOCON parser as part of laika-core, supporting the full spec except for file includes (for now).

- Enhancement for the DocumentTree API

  - The result of a tree parsing operation is now a new type called `DocumentTreeRoot`

  - It has a `coverDocument` property and contains the recursive tree structure of the parsed content.

  - Each `DocumentTree` in the structure now has an explicit `titleDocument: Option[Document]` property for more explicit content organization in e-books.

  - Properties that previously held references to streams and other impure data had been removed from the pure content model (e.g. `DocumentTree.staticDocuments`).

- Bug fixes for fenced code blocks with blank lines in GitHub-Flavored Markdown

0.11.0 (June 12, 2019)

- New Renderer for producing EPUB containers

- New `laikaEPUB` task in the sbt plugin

- New `laikaIncludeEPUB` setting for the `laikaSite` task

- Support for cover images for EPUB and PDF

- Support for document metadata (author, language, date, etc.) for EPUB and PDF

- Support for title pages per chapter

- Backwards-compatible to 0.9.0 and 0.10.0 - if you update from earlier version, please see the release notes for 0.9.0 for migration

0.10.0 (Dec 1, 2018)

- Add support for GitHub Flavored Markdown:

- Tables

- Fenced Code Blocks

- Auto-Links

- Strikethrough

- Preparing for Scala 2.13

  - Adjust use of Collection API for breaking changes and deprecations in 2.13

  - Replace use of parallel collections with custom executor

- Level of parallelism of transformations is now configurable

0.9.0 (Sep 15, 2018)

- New ExtensionBundle APIs allow to bundle extensions into a single object for easier reuse. Supported extension hooks include directives, markup parser extensions, rewrite rules, custom renderers, document type matchers, alternative parsers for stylesheets, templates or configuration headers and default templates per output format.

- Reduced number of settings and tasks in the sbt plugin, by using the new ExtensionBundle API for sbt settings.

- Improved package structure to reduce number of required imports.

0.8.0 (June 4, 2018)

- Doubles parsing speed for both Markdown and reStructuredText

- Much lower number of parser instance creations on repeated runs

- Performance goals had been achieved through replacing the former Scala SDK parser combinators with a custom, optimized combinator design:

  - Fewer dependent types and base parsers in objects instead of traits, making it easier to freely compose parsers

  - Create parser error messages lazily, as most of them will never be accessed

  - Avoid cost of by-name args in all cases except | combinator

- Add support for size and align options for the image directive in reStructuredText

- Fixes for all bugs known and reported to this point

- Address all deprecation warnings for Scala 2.12

0.7.5 (Dec 30, 2017)

- Support for sbt 1.0

- Laika's sbt plugin is now an AutoPlugin

- Prefixed all task and setting keys to adhere to recommended naming pattern (e.g. laikaGenerate) to avoid name conflicts for autoImports

- Adjustments for API changes in sbt 1.0

- Bug fixes in the library

- Drop support for sbt 0.13 and Scala 2.10

0.7.0 (April 17, 2017)

- Support for Scala 2.12 (with continued support for 2.11 and 2.10)

- Redesign and cleanup of the Document API: use case classes wherever possible, extract features into pluggable traits and introduce a new `Cursor` type for tree rewriting

- Allow to customize the `FopFactory` for the PDF renderer (in API and sbt plugin)

- Fix an issue in the `laika:site` task in the sbt plugin that executed several sub-tasks twice which could lead to IllegalStateExceptions caused by the resulting race condition

- Fixes for the reStructuredText parser (for option lists and IP addresses)

0.6.0 (May 23, 2016)

- Support for rendering PDF documents

- Support for rendering XSL-FO output

- New CSS parser supporting a large subset of standard CSS

- Support styling of PDF documents with CSS

- Support for different templates per output format

- New sbt tasks: `html`, `pdf`, `xslfo`, `prettyPrint` for rendering a single output format

- New sbt task `generate` for rendering one or more output formats (e.g. `laika:generate html pdf`)

- Integrate PDF output into existing sbt task `laika:site` via new setting `includePDF`

- New directives `pageBreak`, `style` and `format`

- Changes to the `Render` and `Transform` API to allow for the merging of an entire directory of input files into a single output file (as required by PDF rendering)

0.5.1 (Oct 10, 2015)

- Cross-compile for Scala 2.11 and 2.10

- Publish the sbt plugin to the new plugin repository on Bintray

- Upgrade to ScalaTest 2.2.4

0.5.0 (Jan 9, 2014)

- New sbt plugin, exposing all Laika features and customization hooks as sbt tasks and settings

- New option to merge multiple input directories into a tree structure with a single root, allowing to keep reusable styles or templates ("themes") separately

- New option to use Markdown and reStructuredText markup in the same input tree, including cross-linking between the two formats

- Move to a multi-project build and rename the main artifact from `laika` to `laika-core`

- Upgrade to ScalaTest 2.0 and sbt 0.13

- Drop support for Scala 2.9.x

0.4.0 (Nov 22, 2013)

- Template-based site generation

- Support for tables of contents, convenient cross-linking between documents and autonumbering of documents and sections for all supported markup formats

- Custom Directives for templates and text markup

- Document Fragments that can be rendered separately from the main document content

- New API for batch processing for parse, render or full transform operations

- Parallel processing of parsers and renderers

0.3.0 (Aug 3, 2013)

- Support for most of the standard directives and text roles of the reStructuredText reference parser (admonitions, `figure`, `image`, `code`, `raw` and many more)

- Now integrates the official Markdown test suite (and many fixes to make it pass)

- Now integrates a test for transforming the full reStructuredText specification (which, of course, is written in reStructuredText) and many fixes to make it pass

- Adds the renderer option `HTML.unformatted` for terse output without indentation or whitespace (often desirable when writing the rendered document to a database for example)

- Adds a new Demo App to try out Laika online

- General cleanup of parser implementations and alignments between Markdown and reStructuredText parsers, making both of them much more robust

0.2.0 (May 7, 2013)

- Support for reStructuredText (full specification)

- Concise and type-safe API for all reStructuredText extensibility options (directives, text roles)

- New document tree nodes for tables, footnotes, citations, definition lists, internal links, comments, system messages, invalid elements

- Render hints for document tree nodes in the form of the new Customizable trait

0.1.0 (Jan 12, 2013)

- Support for Markdown as input

- Support for HTML and AST as output

- Customization hooks for renderers

- Document tree rewriting

- Various options for input and output (strings, files, java.io.Reader/Writer, java.io streams)

- Generic base traits for markup parser implementations

Migration Guide

Laika has not reached the 1.0 status yet and as a consequence, minor API changes can still be expected on the road towards 1.0.

Most of the content in this guide is not relevant if you either use the integrated sbt plugin or the library API without any lower-level customizations.

The only change of the top-level library API in 8 years had been the 0.12.0 release, which has very detailed migration instructions below.

Versions older than 0.16.0

The 0.16 release introduced theme support and as a consequence all parsers, renderers and transformers are now provided as a cat-effect `Resource`.

If you are using the sbt plugin, you are not affected by this API change. When using the library API the necessary code changes are quite trivial. Change:

```
val transformer = ??? // build your transformer like before

transformer
  .fromDirectory("docs")
  .toDirectory("target")
  .transform
```

to

```
val transformer = ??? // build your transformer like before

transformer.use {
  _.fromDirectory("docs")
   .toDirectory("target")
   .transform
}
```

Ensure that you create the transformer only once at application start to avoid unnecessary overhead and memory usage for repeated use.

Finally, the legacy directive syntax which had been deprecated in the 0.12 release, has now been removed.

Versions older than 0.15.0

If you plan to migrate to 0.15 it is recommended to address all deprecation warnings the compiler admits as this is the last release where they'll be supported.

This version introduces namespaces for the built-in substitution variables to avoid name clashes with user-defined variables. If you use them in your markup files or templates you need to adjust them for this release:

- `document.*` becomes `cursor.currentDocument.*`

- All configuration variables are now prefixed with `laika`, e.g. `laika.title`, `laika.navigationOrder`.

- `cursor.*` and `laika.*` are the only reserved namespaces, any other prefix can be freely chosen.

If you upgrade from a version older than 0.12.0 it is also recommended to switch to the new directive syntax described below as this is a functional area that unfortunately does not emit any compiler warnings.

Versions older than 0.14.0

- If you use the document AST: the `Path` API had been split into `Path` for absolute paths only and `RelativePath` for pointing from one document to another.

- If you developed parser extensions there are new entry points for defining a span or block parser. You can address all deprecation warnings you might get for your parser definitions by following the new guide for **Writing Parser Extensions**.

Versions older than 0.12.0

The 0.12 release contains the most significant number of breaking changes of any release since the 0.1 version in 2012.

Using Laika's sbt plugin

- If you do not use any of the customization hooks you can safely skip this guide.

- If you define custom rewrite rules, render functions or directives, please see the corresponding sections below.

Using Laika's Library API

- It's probably safest you read all the sections of this guide as even the high level APIs had changed.

- You can of course simply skip features you do not use.

This section is written for users of the 0.9, 0.10 and 0.11 releases which had been compatible with each other.

If you migrate from an even earlier version, please start with the **release notes for 0.9** first.

Motivation

The following design goals led to the partial rewrite:

- Make the library a good citizen in a pure FP setup

- Stop throwing any exceptions in public APIs

- Stop hiding the side-effecting nature of some operations in blocking, synchronous APIs

- Build on top of cats-effect so that users get the familiar Bring-Your-Own Effect setup

- Avoid any kind of runtime reflection

- Make the feature set which had grown organically over the years more consistent and convenient to use

- Do many changes in one big, breaking release so that the road towards a final 1.0 release will be less bumpy from now on

New Separate laika-io Module

A subset of the existing `laika-core` module has been extracted into its own module, `laika-io`. You need to add this new dependency if you use:

- File/Stream IO

- EPUB output

- Parallel transformations

The existing PDF support already had its own, separate artifact `laika-pdf` which is also still available.

If you mostly do in-memory transformations from Markdown or reStructuredText to HTML you can continue using the `laika-core` module.

The reasons for the split were:

- Keep features that depend on cats-effect separate so that the core module does not need this dependency

- Narrow the feature set of the core module to a point where it can realistically be fully supported for Scala.js in a later release (most likely 0.13)

Referential Transparency

Laika is now supposed to be a good citizen in a pure FP setup when using its library API. The following changes had been implemented for this purpose:

- No method in the public API throws any Exceptions anymore.

- The result of pure operations is provided by instances of `Either`.

- The result of side-effecting operations is provided by a return type of `F[A]` where `F[_]` is the standard Bring-Your-Own-Effect pattern, so that Laika can be used with cats-IO, Monix or Zio.

- The dependency on the Typesafe Config library and its impure Java API has been removed in favor of a new lightweight and pure HOCON parser provided by Laika, supporting the full spec.

Parser, Renderer and Transformer APIs

The changes were necessary for the following reasons:

- The reduced core module does no longer offer any IO-related functionality.

- The return types of many methods had to change as Laika does no longer throw any Exceptions.

- The new IO module has its own builder APIs for constructing instances that work with the cats-effect type classes.

It's probably easiest to just show a few before/after examples below:

Transforming from Markdown to HTML in-memory

Before

```scala
val input = "some *text* example"

val transformer = Transform
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)

val res: String = transformer
  .fromString(input)
  .toString
```

After

```scala
val input = "some *text* example"

val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)
  .build

val res: Either[ParserError, String] = transformer
  .transform(input)
```

Transforming a directory of files from Markdown to HTML

Before

```
val transformer = Transform
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)

val res: Unit = transformer
  .fromDirectory("src")
  .toDirectory("target")
```

After (ensure you added the new `laika-io` dependency):

```
import laika.io.implicits._

implicit val cs: ContextShift[IO] =
  IO.contextShift(ExecutionContext.global)

val blocker = Blocker.liftExecutionContext(
  ExecutionContext.fromExecutor(Executors.newCachedThreadPool())
)

val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .using(GitHubFlavor)
  .io(blocker)
  .parallel[IO]
  .build

val res: IO[Unit] = transformer.use {
  _.fromDirectory("src")
   .toDirectory("target")
   .transform
}
```

Note that while the new code sample looks more verbose, it now gives you full control over where your effects are run.

Customizing Renderers

The renderer API had changed from a side-effecting API to a pure API. Renderers now take AST elements and produce a string and will be invoked recursively as before. This model allows for easier testing and pure code at a minimal performance hit over the old design (< 10%).

Before

```
val transformer = Transform
  .from(Markdown)
  .to(HTML)
  .rendering { out =>
    { case Emphasized(content, _) =>
      out << """<em class="big">""" << content << "</em>" }
  }
```

After

```
val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .rendering {
    case (fmt, Emphasized(content, opt)) =>
      fmt.element("em", opt, content, "class" -> "big")
    }.build
```

The engine passes the following instances to the partial function:

- The formatter (`fmt`) which has a different API for each supported output format, in this case it will be an `HTMLFormatter`

- The current element of the rendered document AST which you can pattern match on

Rewrite Rules

Here most of the changes are in the implementation which had been rewritten to avoid any kind of runtime reflection.

On the API surface, there are only two changes:

- The return type is more explicit (e.g. `Replace(newElement)` instead of `Some(newElement)`)

- Rules for rewriting spans and blocks get registered separately for increased type-safety, as it is invalid to replace a span with a block element.

Before

```
val transformer = Transform
  .from(Markdown)
  .to(HTML)
  .usingRule {
    case Emphasized(content, opts) => Some(Strong(content, opts))
  }
```

After

```
val transformer = Transformer
  .from(Markdown)
  .to(HTML)
  .usingSpanRule {
    case Emphasized(content, opts) => Replace(Strong(content, opts))
  }
  .build
```

Templating

Variable substitutions now use HOCON syntax in line with the general deeper integration with HOCON throughout Laika's feature set.

The old syntax is still supported, but will be removed at some point before the 1.0 release.

The old reference style `{{some.ref}}` now becomes either `${some.ref}` for a required reference or `${?some.ref}` for an optional one.

Directives

Directives in 0.12 come with changes in both, the DSL for creating custom directives and the supported syntax in markup and template files.

If you are not implementing your own directives, but only use the built-in ones provided by Laika, you can skip the section on the DSL.

Directive Syntax

- The separators for the attribute and body sections have changed

- HOCON syntax is now used for named attributes between curly braces or a plain string for an unnamed attribute in parenthesis

- The old syntax is still supported, but will be removed at some point before the 1.0 release

Before

```
@:if "layout.showSidebar":
  <div class="sidebar">...</div>

~else:
  <p>Something else</p>
```

After

```
@:if(layout.showSidebar)
<div class="sidebar">...</div>

@:else
<p>Something else</p>

@:@
```

Directive DSL

The building blocks for creating your own directives have also changed significantly:

- `attribute(Default)` is now `attribute(0)`

- `body` is now either `parsedBody` or `rawBody`

- Type conversions happen with the new `as` method: `attribute("title").as[String]`, based on the `ConfigDecoder` type class that is also used for the new Config API

- Named body parts have been replaced by the more flexible **Separator Directives**.

- The built-in helper for mapping directive parts with different arity has been replaced by cats `mapN`

Before

```
import laika.ast._
import Spans.dsl._

case class Note (title: String,
                 content: Seq[Span],
                 options: Options = NoOpt) extends Span
                                           with SpanContainer[Note]

val spanDirective = Spans.create("note") {
  (attribute(Default) ~ body(Default)) (Note(_,_))
}
```

After

```
import cats.implicits._
import laika.ast._
import Spans.dsl._

case class Note (title: String,
                 content: Seq[Span],
                 options: Options = NoOpt) extends Span
                                           with SpanContainer[Note]

val spanDirective = Spans.create("note") {
  (attribute(0).as[String], parsedBody).mapN(Note(_,_))
}
```

Note the additional import of cats implicits in the new version.

Config API

Laika's support for HOCON configuration, originating either from configuration headers in markup or template documents or separate configuration files had previously been based on the Typesafe Config library and its Java API. Instances of `Config` were available in the `Document` and `DocumentTree` types.

This type is now Laika's own Config API, based on its own lightweight HOCON parser. See the API docs for **laika.config.Config** for details.

Document Tree Model

The model had been enhanced to better cater for Laika's support for e-book generation.

- The result of a tree parsing operation is now a new type called `DocumentTreeRoot`

- It has a `coverDocument` property and contains the recursive tree structure of the parsed content.

- Each `DocumentTree` in the structure now has an explicit `titleDocument:   Option[Document]` property for more explicit content organization in e-books.

- Properties that previously held references to streams and other impure data had been removed from the pure content model (e.g. `DocumentTree.staticDocuments`).