Scalite

A blog-aware static site generator

by M Ahsan Al Mahir

on February 16, 2023

What is it?

Scalite is a general purpose static site generator.

Inputs are:

- Contents of the website
  - in the form of formatted text files such as Markdown, reStructuredText, Textile etc.

Scalite is a general purpose static site generator.

Inputs are:

- Contents of the website
    - in the form of formatted text files such as Markdown, reStructuredText, Textile etc.
- Templates specifying the layout of the webpages
    - rendered with the contents of text files and other parameters

Scalite is a general purpose static site generator.

Inputs are:

- Contents of the website
  - in the form of formatted text files such as Markdown, reStructuredText, Textile etc.
- Templates specifying the layout of the webpages
  - rendered with the contents of text files and other parameters
- Configuration files
  - YAML, JSON files specifying configurations, variables etc.

Scalite is a general purpose static site generator.

Inputs are:

- Contents of the website
  - in the form of formatted text files such as Markdown, reStructuredText, Textile etc.
- Templates specifying the layout of the webpages
  - rendered with the contents of text files and other parameters
- Configuration files
  - YAML, JSON files specifying configurations, variables etc.
- Stylesheets: SASS, CSS etc.

Scalite is a general purpose static site generator.

Inputs are:

- Contents of the website
    - in the form of formatted text files such as Markdown, reStructuredText, Textile etc.
- Templates specifying the layout of the webpages
    - rendered with the contents of text files and other parameters
- Configuration files
    - YAML, JSON files specifying configurations, variables etc.
- Stylesheets: SASS, CSS etc.

The output is a static website with the HTML, CSS and JS files in proper directory structure.

» How it works

The user sets up a folder with the following folder structure:

```
.
+-- _assets
|   +-- website assets go here
+-- _layouts
|   +-- template files go here
+-- _plugins
|   +-- plugins go here
+-- _posts
|   +-- posts (in a blog) or other page contents go here
+-- _sass
|   +-- stylesheets go here
+-- _config.yml
```

» How it works

Scalite then finds all the files, loads them in the runtime. It then

- Reads the configurations
    - Loads the specified plugins

» How it works

Scalite then finds all the files, loads them in the runtime. It then

- Reads the configurations
  - Loads the specified plugins
- Converts all formatted text files into HTML files
  - Users can provide converter plugins to support new formats

» How it works

Scalite then finds all the files, loads them in the runtime. It then

- Reads the configurations
  - Loads the specified plugins
- Converts all formatted text files into HTML files
  - Users can provide converter plugins to support new formats
- Create `Page` objects, and
  - handles cross referencing
  - creates categories and tags
  - and other features the user specifies

» How it works

Scalite then finds all the files, loads them in the runtime. It then

- Reads the configurations
  - Loads the specified plugins
- Converts all formatted text files into HTML files
  - Users can provide converter plugins to support new formats
- Create `Page` objects, and
  - handles cross referencing
  - creates categories and tags
  - and other features the user specifies
- For each specified page, loads its template, converted contents and compiles the files into final HTML files

» How it works

Scalite then finds all the files, loads them in the runtime. It then

- Reads the configurations
  - Loads the specified plugins
- Converts all formatted text files into HTML files
  - Users can provide converter plugins to support new formats
- Create `Page` objects, and
  - handles cross referencing
  - creates categories and tags
  - and other features the user specifies
- For each specified page, loads its template, converted contents and compiles the files into final HTML files
- Creates the destination folder, copies the pages and assets to the destination folder

Listing: hello_world.md

```
---
# YAML header
# local variables, configs
title: Front Page
tag: tag1, tag2
---

Hello, **World**!
```

Listing: page.mustache

```
<html>
  <body>
    <header>
      {{ title }}
    </header>
    {{> content}}
  </body>
</html>
```

Listing: hello_world.html

```
<html>
  <body>
    <header>
      Front Page
    </header>
    Hello, <b>World</b>!
  </body>
</html>
```

» Motivation

Scalite is inspired from **Jekyll**. It's is an attempt to recreate Jekyll while generalizing many of its features.

Scalite attempts to identify the core components, and make them as loosely coupled as possible.

» Motivation

Scalite's core goals are:

- To be language agnostic:

» Motivation

Scalite's core goals are:

- To be language agnostic:
    - it should support contents written in any markup language

» Motivation

Scalite’s core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language

» Motivation

Scalite's core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language
- To be easy to use, configurable, and flexible

» Motivation

Scalite's core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language
- To be easy to use, configurable, and flexible
  - all the features should be customizable through config files

» Motivation

Scalite's core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language
- To be easy to use, configurable, and flexible
  - all the features should be customizable through config files
- To be extensible and fully customizable

» Motivation

Scalite's core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language
- To be easy to use, configurable, and flexible
  - all the features should be customizable through config files
- To be extensible and fully customizable
  - plugins should have a powerful API

» Motivation

Scalite’s core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language
- To be easy to use, configurable, and flexible
  - all the features should be customizable through config files
- To be extensible and fully customizable
  - plugins should have a powerful API
- It should never assume any particular website structure

» Motivation

Scalite's core goals are:

- To be language agnostic:
  - it should support contents written in any markup language
  - it should support any templating language
- To be easy to use, configurable, and flexible
  - all the features should be customizable through config files
- To be extensible and fully customizable
  - plugins should have a powerful API
- It should never assume any particular website structure
  - the website structure should be easily modifyable

» Sacrifices

However, flexibility comes at a cost,

- Implementation becomes exponentially complex

» Sacrifices

However, flexibility comes at a cost,

- Implementation becomes exponentially complex
    - strong type system of Scala makes it more complicated

» Sacrifices

However, flexibility comes at a cost,

- Implementation becomes exponentially complex
    - strong type system of Scala makes it more complicated
- Class definitions are generalized to be loaded at runtime to simplify plugin implementation

» Sacrifices

However, flexibility comes at a cost,

- Implementation becomes exponentially complex
    - strong type system of Scala makes it more complicated
- Class definitions are generalized to be loaded at runtime to simplify plugin implementation
    - But this incurs heavy runtime overhead

» Sacrifices

However, flexibility comes at a cost,

- Implementation becomes exponentially complex
  - strong type system of Scala makes it more complicated
- Class definitions are generalized to be loaded at runtime to simplify plugin implementation
  - But this incurs heavy runtime overhead
- Great deal of consideration is needed to define what plugins can and can’t control

» Sacrifices

However, flexibility comes at a cost,

- Implementation becomes exponentially complex
    - strong type system of Scala makes it more complicated
- Class definitions are generalized to be loaded at runtime to simplify plugin implementation
    - But this incurs heavy runtime overhead
- Great deal of consideration is needed to define what plugins can and can’t control
    - There might be data leakage bugs unless immutable data structures are used throughout

Project Structure

» Project structure: Basis

Basis modules that are more or less used by all other modules:

- **`documents`**: module defining interfaces for common features
  - **`Assets`**: Class to handle asset files
  - **`Convertible`**: files that need to be converted to some other format
  - **`Page, Renderable, SourceFile`**: Mixins defining properties

» Project structure: Basis

Basis modules that are more or less used by all other modules:

- **`documents`**: module defining interfaces for common features
  - **`Assets`**: Class to handle asset files
  - **`Convertible`**: files that need to be converted to some other format
  - **`Page`, `Renderable`, `SourceFile`**: Mixins defining properties
- **`data`**: module defining JSON-like data structures
  - **`immutable`**: read-only JSON-like structures.
    - communicating with external plugins
    - storing global variables
  - **`mutable`**: mutable JSON-like structures

» Project structure: Basis

Basis modules that are more or less used by all other modules:

- `documents`: module defining interfaces for common features
  - `Assets`: Class to handle asset files
  - `Convertible`: files that need to be converted to some other format
  - `Page`, `Renderable`, `SourceFile`: Mixins defining properties
- `data`: module defining JSON-like data structures
  - `immutable`: read-only JSON-like structures.
    - communicating with external plugins
    - storing global variables
  - `mutable`: mutable JSON-like structures
- `util`: utility functions, parsers, logging mechanism etc.

» Project structure: Convertibles

- **`converters`**: module defining source file -> html logic

» Project structure: Convertibles

- `converters`: module defining source file -> html logic
  - `Converter`: defines `Converter` interface

» Project structure: Convertibles

- **`converters`**: module defining source file -> html logic
  - **`Converter`**: defines **`Converter`** interface
  - **`Converters`**: singleton object contianing all converters available at runtime

» Project structure: Convertibles

- `converters`: module defining source file -> html logic
  - `Converter`: defines `Converter` interface
  - `Converters`: singleton object contianing all converters available at runtime
- `layouts`: module defining template files

» Project structure: Convertibles

- `converters`: module defining source file -> html logic
  - `Converter`: defines `Converter` interface
  - `Converters`: singleton object contianing all converters available at runtime
- `layouts`: module defining template files
  - `Layout`: Interface for a single template

» Project structure: Convertibles

- **`converters`**: module defining source file -> html logic
  - **`Converter`**: defines **`Converter`** interface
  - **`Converters`**: singleton object contianing all converters available at runtime
- **`layouts`**: module defining template files
  - **`Layout`**: Interface for a single template
  - **`LayoutGroup`**: Set of templates of the same language

» Project structure: Convertibles

- `converters`: module defining source file -> html logic
  - `Converter`: defines `Converter` interface
  - `Converters`: singleton object contianing all converters available at runtime
- `layouts`: module defining template files
  - `Layout`: Interface for a single template
  - `LayoutGroup`: Set of templates of the same language
    - custom `LayoutGroups` for other languages via plugins

» Project structure: Convertibles

- **`converters`**: module defining source file -> html logic
  - **`Converter`**: defines **`Converter`** interface
  - **`Converters`**: singleton object contianing all converters available at runtime
- **`layouts`**: module defining template files
  - **`Layout`**: Interface for a single template
  - **`LayoutGroup`**: Set of templates of the same language
    - custom **`LayoutGroup`**s for other languages via plugins
  - **`Layouts`**: Singleton object with all layouts of various languages available at runtime

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
  - **`Element`**: interface of an element

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
    - **`Element`**: interface of an element
        - **`PostLike`**: blog-post like contents

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
  - **`Element`**: interface of an element
    - **`PostLike`**: blog-post like contents
    - **`PageLike`**: static page like content, i.e. **`about, index`** pages

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
  - **`Element`**: interface of an element
    - **`PostLike`**: blog-post like contents
    - **`PageLike`**: static page like content, i.e. **`about, index`** pages
    - **`ItemLike`**: non-page elements to be inserted in other elements

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
  - **`Element`**: interface of an element
    - **`PostLike`**: blog-post like contents
    - **`PageLike`**: static page like content, i.e. **`about, index`** pages
    - **`ItemLike`**: non-page elements to be inserted in other elements
    - **`user_defined`**: custom element styles via plugins

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
    - **`Element`**: interface of an element
        - **`PostLike`**: blog-post like contents
        - **`PageLike`**: static page like content, i.e. **`about, index`** pages
        - **`ItemLike`**: non-page elements to be inserted in other elements
        - **`user_defined`**: custom element styles via plugins

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
  - **`Element`**: interface of an element
    - **`PostLike`**: blog-post like contents
    - **`PageLike`**: static page like content, i.e. **`about, index`** pages
    - **`ItemLike`**: non-page elements to be inserted in other elements
    - **`user_defined`**: custom element styles via plugins
  - **`Collection`**: a single collection, typically sourced from a single folder.

» Project Structure: Website content

Contents of the website are defined as `Collections` of `Elements`.

- **`collections`**: collection of contents
  - **`Element`**: interface of an element
    - **`PostLike`**: blog-post like contents
    - **`PageLike`**: static page like content, i.e. **`about, index`** pages
    - **`ItemLike`**: non-page elements to be inserted in other elements
    - **`user_defined`**: custom element styles via plugins
  - **`Collection`**: a single collection, typically sourced from a single folder.
  - **`Collections`**: Singleton object holding all available collections

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages
  - **`Tree`**: Interface defining nodes of a `Tree` containing contents

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages
  - **`Tree`**: Interface defining nodes of a `Tree` containing contents
  - **`AnyTree`**: A generic implementation of `Tree`.

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages
  - **`Tree`**: Interface defining nodes of a `Tree` containing contents
  - **`AnyTree`**: A generic implementation of `Tree`.
    - **`CategoryStyle`**: defines a category style tree structure

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages
  - **`Tree`**: Interface defining nodes of a `Tree` containing contents
  - **`AnyTree`**: A generic implementation of `Tree`.
    - **`CategoryStyle`**: defines a category style tree structure
    - **`TagStyle`**: defines shallow trees for Tags

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages
  - **`Tree`**: Interface defining nodes of a `Tree` containing contents
  - **`AnyTree`**: A generic implementation of `Tree`.
    - **`CategoryStyle`**: defines a category style tree structure
    - **`TagStyle`**: defines shallow trees for Tags
  - **`Forest`**: A set of specific type of `Trees`

» Project structure: Categorizing

In any website, webpages are stored in the device in a tree like structure. Similarly, blogs typically have categories, tags etc. to structure the posts. We generalize this notion using `trees` module:

- **`trees`**: module defining structure of the pages
  - **`Tree`**: Interface defining nodes of a `Tree` containing contents
  - **`AnyTree`**: A generic implementation of `Tree`.
    - **`CategoryStyle`**: defines a category style tree structure
    - **`TagStyle`**: defines shallow trees for Tags
  - **`Forest`**: A set of specific type of `Trees`
  - **`PostForests`**: Singleton object holding all `Forests`

» Project structure: Categorizing

```
        tags
 ________|________
|        |        |
tag1   tag4     tag5
```

```
              categories
 _____________|_____________
|             |             |
cat1         cat4          cat5
 ___|___                    |
|       |                  cat6
cat2   cat3
```

Here, each post, page or any other element can belong to any number of tag `tag`$_i$ and any number of category `cat`$_i$.

» Project structure: Categorizing

```
          tags                                          categories
   ________|________                          ______________|______________
   |        |        |                        |              |              |
 tag1     tag4     tag5                     cat1           cat4           cat5
                                         ____|____                          |
                                         |        |                       cat6
                                       cat2     cat3
```

Here, each post, page or any other element can belong to any number of tag $\texttt{tag}_i$ and any number of category $\texttt{cat}_i$.

If the user wishes, these tree nodes can be rendered into seperate webpages to make navigation in the website easier.

» Project structure: Plugins and Hooks

- `Plugin`: An interface for objects that can be provided as plugins
    - `PluginManager`: Loads all specified plugins from `/_plugins`

» Project structure: Plugins and Hooks

- **`Plugin`**: An interface for objects that can be provided as plugins
    - **`PluginManager`**: Loads all specified plugins from `/_plugins`
- **`hooks`**: define runtime hooks for fine-grained control.
    - Each module has places where hook plugins can anchor to and modify the data-flow

» Project structure: Plugins and Hooks

- **`Plugin`**: An interface for objects that can be provided as plugins
  - **`PluginManager`**: Loads all specified plugins from `/_plugins`
- **`hooks`**: define runtime hooks for fine-grained control.
  - Each module has places where hook plugins can anchor to and modify the data-flow
  - For example,
    - before or after reading source files,
    - before and after conversion,
    - before or after rending templates etc.

» Project structure: Plugins and Hooks

- **`Plugin`**: An interface for objects that can be provided as plugins
  - **`PluginManager`**: Loads all specified plugins from `/_plugins`
- **`hooks`**: define runtime hooks for fine-grained control.
  - Each module has places where hook plugins can anchor to and modify the data-flow
  - For example,
    - before or after reading source files,
    - before and after conversion,
    - before or after rending templates etc.
  - Hooks are implemented in Publisher-Subscriber pattern

» Project structure: Plugins and Hooks

- **`Plugin`**: An interface for objects that can be provided as plugins
  - **`PluginManager`**: Loads all specified plugins from `/_plugins`
- **`hooks`**: define runtime hooks for fine-grained control.
  - Each module has places where hook plugins can anchor to and modify the data-flow
  - For example,
    - before or after reading source files,
    - before and after conversion,
    - before or after rending templates etc.
  - Hooks are implemented in Publisher-Subscriber pattern
  - They are added to the objects at runtime by the `PluginManager`.

» Project structure: Config file

The config file `/_config.yml` allows the user to set global configurations. It has the structure

```
#-- Global settings and variables --#
title: My site  # variable
show_excerpts: false  # setting

#-- Module settings --#
collections: #-- settings to be passed to Collections object --#
    posts: true
    articles:
        output: true
        folder: /_articles
plugins:
    textile: #-- settings to be passed to textile plugin --#
```

» Project structure: Putting it together

`Site` class at the root of the project puts all of the components together:

- Load global configurations from `/_config.yml`
- Send configurations to each module and load plugins
- `build()` command processes all the modules

» Libraries

- **`weePickle`**: To read configurations from YAML files and create internal repsentations of the data
- **`nscala-time`**: To handle date-time calculation
- **`scala-parallel-collections`**: For concurrency
- **`scala-uri`**: To parse/simplify url/uri
- **`laika`**: For the default Markdown->HTML converter
- **`scala-logging`** and **`logback-classic`**: To support logging
- **`scala-mustache`**: The default compiler for mustache templates

Implementation Details

» Design patterns

- Modules are written in builder design pattern.
  - The constructors for the objects are specified at the runtime, based on the plugins provided, or the configuration files

» Design patterns

- Modules are written in builder design pattern.
    - The constructors for the objects are specified at the runtime, based on the plugins provided, or the configuration files
- Work is done by singleton objects, that
    - receive configurations
    - set up the constructors
    - fetches and arranges files
    - compilers/renders files
    - writes them back to the disk

» Collections

```
object Collections extends Configurable with Generator:
  /** section in the configs */
  val sectionName: String = "collections"

  /** Avaiable Element styles */
  private val styles = LinkedHashMap[String, ElemConstructor](
    "post" -> PostConstructor,
    "page" -> PageConstructor,
    "item" -> ItemConstructor
  )

  /** Plugins may add more constructors to this table */
  def addStyle(elemCons: ElemConstructor): Unit = ...
  ...
```

» Collections: apply

```scala
...
private val collections = ListBuffer[Collection]()

/** process all collections and writes to destination */
def process(dryRun: Boolean = false): Unit =
  for col <- collections.par do col.process(dryRun)

/** Gets configuration set in "collections" section of
  * '_configs.yml' and creates necessary Collection objects */
def apply(_configs: MObj, globals: IObj): Unit =
  // update method is provided by data module
  val configs = defaultConfigs update _configs

  // get values from the configs
  val base = /** base directory **/
  val colsDir = /** relative directory where collections are */
```

» Collections: apply cont.

```
    // create collection for each name in collecionsDir
    for (name, config) <- configs do
      config match
        case config: MObj =>
          // what kind of elements we want to make
          val style = config.extractOrElse("style")("item")
          val output = /** write the elements to destination? */
          if !output then logger.debug(s"won't output ${RED(name)}")
          else
            val dir = /** absolute directory of elements */
            val Col =
              Collection( styles(style), // element constructor
                          name, dir, configs, globals )
            /** handle item collections so that other collections
              * have access to the items */
            collections += Col // add to the collections map
        case _ => /** log error */
```

» Collection

```scala
class Collection(
    private val elemCons: ElemConstructor, // element constructor
    val name: String, // name of collection
    private val directory: String, // absolute directory
    _configs: MObj, // configs for this collection
    protected val globals: IObj // global configs and variables
) extends Renderable with Page:

  /** fetch all items of this collection */
  lazy val items: Map[String, Element] =
    lazy val constructor =  elemCons(name) // create the constructor
    val files = getListOfFilepaths(directory)

    def f(fn: String) = /** construct fn -> Element object pair */
    files.filter(Converters.hasConverter).map(f).toMap
```

» Collection: process

```
protected[collections] def process(dryrun: Boolean = false) =
  for item <- items.values do
    item match
      case item: Page => item.write(dryrun) // only write Pages
      case _          => ()
  write(dryrun) // write the index page of this collection
  CollectionHooks.afterWrites(globals)(this)
  // run after write hooks attached to the collections
```

» Page: write

```scala
trait Page:
  this: Renderable =>

  def write(dryRun: Boolean = false): Unit =
    if !visible then return
    val path = /** destination */

    if !dryRun then
      val up = PageHooks.beforeRenders(globals)(locals)
      val str = render(IObj(up))
      val r = PageHooks.afterRenders(globals)(locals, str)
      writeTo(path, r)
      PageHooks.afterWrites(globals)(this)
    else return
```