Anna: a lightning-fast static site generator written in Go

Adhesh Athrey, Aditya Hegde, Anirudh Sudhir, Nathan Paul

6th July 2024

gh/anna-ssg

Anna is designed to be fast, highly configurable and easy to use.

Currently, anna beats Hugo and Eleventy another widely used static site generator implemented in JavaScript, on several benchmarks

What does a static site generator do?

A Static Site Generator (SSG) merges templates with content files to produce static websites by converting markdown into static HTML files, resulting in fast and easy-to-maintain websites.

Fan art variation of the Go mascot, modified by Nats Romanova. CC BY-3.0

Genesis

The ACM student chapter at PESU ECC conducts an annual six-week **ACM Industrial Experience Program** where students work as a team to develop industrial-level projects. (AIEP (https://acmpesuecc.github.io/aiep/anna)) gave us hands-on experience with real-world projects and is an excellent opportunity to interact with like-minded individuals.

Discussions over a week revealed how tiring it is to maintain blog sites, a task that becomes more cumbersome as content grows. We decided to tackle this problem by developing our own SSG, in Go!

We began the project in a unique manner

Each of us creating our own prototype of the SSG. This was done to familiarize everyone with the Go toolchain.

The first version of the SSG did just **three** things.

- It rendered markdown (stored in files in a content folder in the project directory) to HTML.
- This HTML was injected into a layout.html file and served over a local web server.
- Later, we implemented a front matter YAML parser to retrieve page metadata

What's in a name?

Probably the first thing that the four of us came across when joining ACM and HSP was the famous Saaru repository, another SSG built by one of our seniors. Saaru is a thin lentil soup, usually served with rices

This was just a playful stunt that we engaged in. We planned on beating Saaru at site render times, optimizing at runtime.

```
In Kannada, rice is referred to as (ಅನ್ನ) pronounced: /ɐnːɐ/
```

Starting Point: CLI Development

Our journey began with developing a command-line interface (CLI) for our static site generator. We used the Cobra library to create a robust and user-friendly CLI. Cobra's features allowed us to easily define commands, flags, and commands, making our tool intuitive and efficient for users.

All the available flags and options can be viewed by entering *anna -h*

By leveraging Cobra, we provided a structured and organized interface, enabling users to interact with our static site generator seamlessly. This foundation set the stage for implementing additional features and optimizations.

Profiling

Team utilized the Pprof library as a one-stop solution for profiling and performance analysis. By implementing a code profiling mechanism using Pprof, the team was able to analyze the execution time of the code and identify potential bottlenecks for further optimization. This allowed them to focus on improving the performance of the static site generator and achieve faster rendering times. The Pprof library provided valuable insights into the code's execution, enabling the team to make informed decisions and optimize the performance of the project effectively.

Basic implementation

```go
func (cmd *Cmd) PrintStats(elapsedTime time.Duration) {
    memStats := new(runtime.MemStats)
    runtime.ReadMemStats(memStats)
    log.Printf("Memory Usage: %d bytes", memStats.Alloc)
    log.Printf("Time Elapsed: %s", elapsedTime)
    cpuUsage := runtime.NumCPU()
    threads := runtime.GOMAXPROCS(0)
    runtime.ReadMemStats(memStats)

    log.Printf("Threads: %d", threads)
    log.Printf("Cores: %d", cpuUsage)
    log.Printf("Time Taken: %s", elapsedTime)
    log.Printf("Allocated Memory: %d bytes", memStats.Alloc)
    log.Printf("Total Memory Allocated: %d bytes", memStats.TotalAlloc)
    log.Printf("Heap Memory In Use: %d bytes", memStats.HeapInuse)
    log.Printf("Heap Memory Idle: %d bytes", memStats.HeapIdle)
    log.Printf("Heap Memory Released: %d bytes", memStats.HeapReleased)
    log.Printf("Number of Goroutines: %d", runtime.NumGoroutine())

    // Get the function with the highest CPU usage
    pc, _, _, _ := runtime.Caller(1)
    function := runtime.FuncForPC(pc)
    log.Printf("Function with Highest CPU Usage: %s", function.Name())
}
```

Directory Watchdog and Live Render

We implemented a directory watchdog system to streamline our workflow, avoiding the need to manually restart the program for code or content changes. This system monitors the working directory for modifications, automatically triggering a rerendering process and updating the output in real-time. Additionally, it serves the rendered content for easy previewing, allowing us to see changes immediately. This improvement has made our development process more efficient and seamless, enabling quick iterations without manual intervention.

POC

We implemented a live reload system using a liveReload struct to enhance our development workflow. The struct maintains an error logger, a map of file modification times, directories to monitor, file extensions to watch, a server status flag, and the site data path.

```
type liveReload struct {
    errorLogger   *log.Logger
    fileTimes     map[string]time.Time
    rootDirs      []string // Directories to monitor, so add or remove as needed
    extensions    []string // File extensions to monitor
    serverRunning bool
    siteDataPath  string
}
```

The system scans specified directories for changes in files with given extensions. When a change in fileTimes is detected, it triggers the LiveReload pipeline to rerender content and update the output. This ensures real-time previews without manual intervention, enhancing development efficiency.

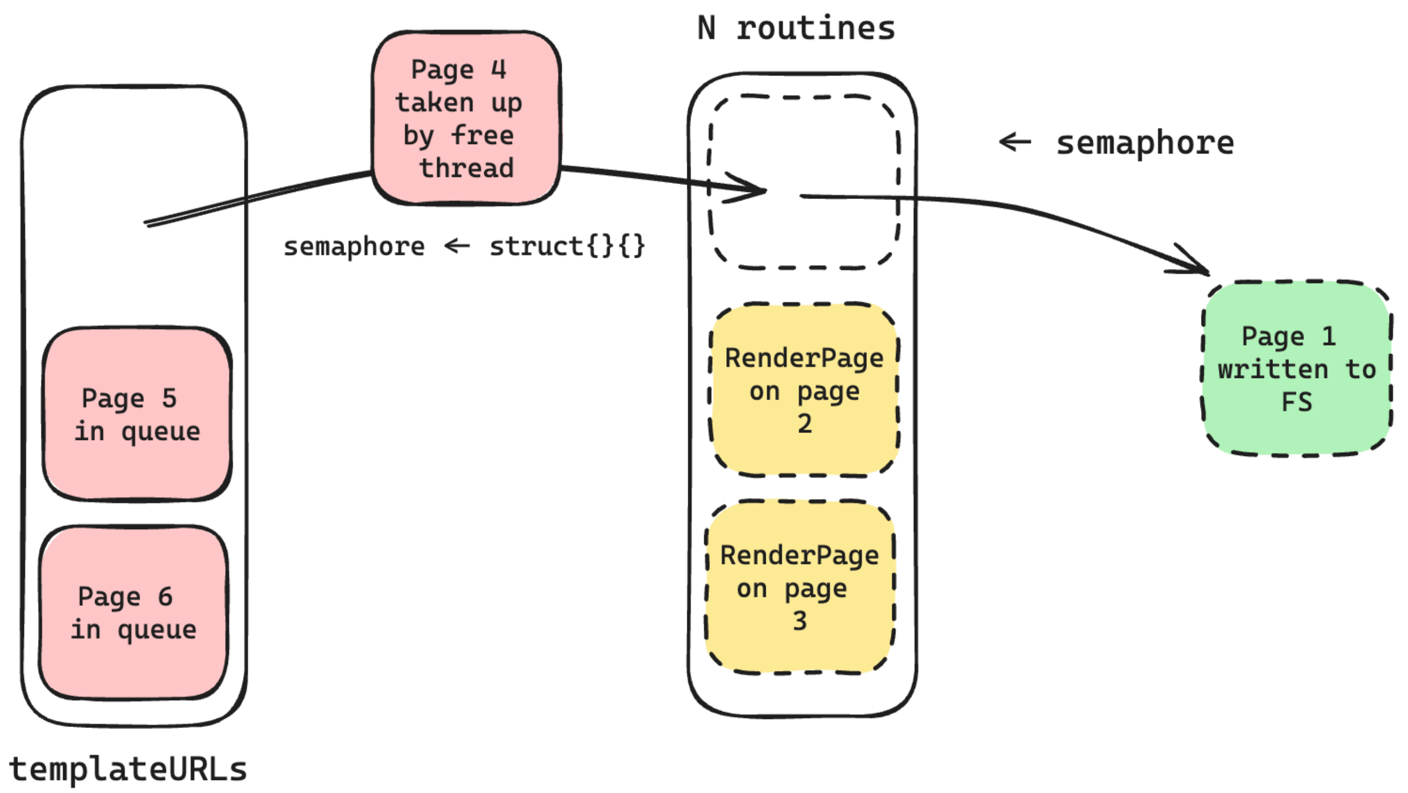
Parallel Rendering pipelines

To build a fast and efficient static site generator , we implemented parallel rendering using goroutines. Instead of rendering Markdown files iteratively, we utilized goroutines to process them concurrently.

By spawning a goroutine for each Markdown file, we significantly reduced the rendering time. This approach leverages Go's concurrency model, allowing multiple files to be rendered simultaneously, thus improving overall performance.

Curent Parallel Rendering Pipeline Model

```
N := numCPU * 2
if numTemplates < N {
    N = numTemplates
}
```



Curent Parallel Rendering Pipeline Model

```go
func (e *Engine) RenderUserDefinedPages(fileOutPath string, templates *template.Template) {
    numTemplates := len(e.DeepDataMerge.Templates)
    concurrency := numCPU * 2 // Simple way to keep limit on concurrency based on System Resources
    if numTemplates < concurrency {
        concurrency = numTemplates
    }
    var wg sync.WaitGroup
    semaphore := make(chan struct{}, concurrency)
    for _, templateURL := range templateURLs {
        if templateURL == ".html" {
            continue
        }
        wg.Add(1)
        semaphore <- struct{}{}
        go func(templateURL string) {
            defer func() {
                <-semaphore
                wg.Done()
            }()
            e.RenderPage(fileOutPath, template.URL(templateURL), templates,
                e.DeepDataMerge.Templates[template.URL(templateURL)].Frontmatter.Layout)
        }(templateURL)
    }
    wg.Wait()
}
```

Future Optimizations Plans

To further enhance performance, future plans include:

Fine-Tuning the Parallel Rendering Pipeline:

- Refine our current parallel rendering process to achieve even better efficiency and speed.

Switching to bufio:

- Utilize bufio for more efficient buffered I/O operations, replacing regular I/O methods.

Caching Frequently Used Files:

- Implement caching for frequently accessed files, such as templates, to reduce load times.

Future Optimizations Plans

Worker Pool Implementation:

- Introduce a worker pool to manage concurrent rendering tasks more effectively.

File Read-Ahead:

- Implement file read-ahead strategies to minimize idle time by preloading necessary data.

Optimizing I/O Operations:

- Streamline other I/O processes to reduce the number of syscalls, thereby improving overall performance.

Benchmarking and Stress Testing

With all current optimizations and future plans in place, we recognized the need for a comprehensive benchmarking and stress testing strategy to evaluate our progress and compare it against competitors. This involves:

Base Performance Metrics:

- Measure key performance indicators (KPIs) such as render time, memory usage, and CPU utilization for our static site generator with the help of the profiling system already set up.

Creating Benchmarking Scenarios:

- Develop a variety of test cases, including different site sizes, complexity levels, and types of content, to assess performance under various conditions.

Benchmarking and Stress Testing

Stress Testing:

- Conduct stress tests to push the system to its limits and identify bottlenecks by rendering thousands of Markdown content pages. This helps in understanding how the system performs under heavy load and what areas need improvement.

Comparative Analysis:

- Benchmark our static site generator against competitors, using the same test scenarios, to see where we stand and to identify areas for possible optimization.

Continuous Monitoring and Feedback:

- With the tools implemented so far, such as Live Render and Profiling, we can achieve continuous monitoring during the benchmarking process. These tools provide real-time feedback and detailed reports on performance metrics.

Benchmark Data and Tested Cases and Related Information

- We have performed benchmark runs comparing our ssg's performance with others such as Hugo (https://gohugo.io), Eleventy (https://www.11ty.dev/), Saaru (https://saaru-docs.netlify.app/) and Sapling (https://sapling.navinshrinivas.com/)

```
Benchmark 1: cd /tmp/bench/11ty && npx @11ty/eleventy
  Time (mean ± σ):      1.297 s ±  0.024 s    [User: 1.414 s, System: 0.308 s]
  Range (min … max):    1.257 s …  1.338 s    10 runs

Benchmark 2: cd /tmp/bench/hugo && hugo
  Time (mean ± σ):     521.7 ms ±   9.2 ms    [User: 1460.5 ms, System: 184.5 ms]
  Range (min … max):   505.0 ms … 535.5 ms    10 runs

Benchmark 3: cd /tmp/bench/anna && ./anna -r "site/"
  Time (mean ± σ):     270.3 ms ±   2.9 ms    [User: 356.2 ms, System: 149.9 ms]
  Range (min … max):   266.8 ms … 275.3 ms    10 runs

Benchmark 4: cd /tmp/bench/saaru && ./saaru --base-path ./docs
  Time (mean ± σ):     143.9 ms ±   1.3 ms    [User: 244.5 ms, System: 111.4 ms]
  Range (min … max):   141.9 ms … 146.2 ms    19 runs

Benchmark 5: cd /tmp/bench/sapling/benchmark && ./../sapling run
  Time (mean ± σ):     220.5 ms ±   5.2 ms    [User: 211.1 ms, System: 199.4 ms]
  Range (min … max):   209.8 ms … 227.7 ms    12 runs
```

Benchmark Data and Tested Cases and Related Information

```
Summary
  'cd /tmp/bench/saaru && ./saaru --base-path ./docs' ran
    1.53 ± 0.04 times faster than 'cd /tmp/bench/sapling/benchmark && ./../sapling run'
    1.88 ± 0.03 times faster than 'cd /tmp/bench/anna && ./anna -r "site/"'
    3.63 ± 0.07 times faster than 'cd /tmp/bench/hugo && hugo'
    9.02 ± 0.18 times faster than 'cd /tmp/bench/11ty && npx @11ty/eleventy'
```

RSS / Atom feed

RSS and Atom feeds are standardized formats used for distributing and accessing updates from websites. They provide structured metadata such as titles, links, and publication dates, allowing users and applications to efficiently stay informed about new content without visiting each site individually

You get to read articles offline and cache posts for later use. You can follow websites made with anna inside Brave (a popular web browser), Thunderbird or newsboat like I do.

Users can have blog sites of their own and do not have to worry about distribution or managing newsfeeds. Feeds in anna are turned on by default.

A buffer is used to efficiently build the XML content before writing it to the file.

This avoids multiple file writes, which is slower.

```
func (e *Engine) GenerateFeed() {
    var buffer bytes.Buffer
}
```

Adding XML declaration and initializing root elements in buffer.

```
buffer.WriteString("<?xml version=\"1.0\" encoding=\"utf-8\" standalone=\"yes\"?>\n")
buffer.WriteString("<?xml-stylesheet href=\"/static/styles/feed.xsl\" type=\"text/xsl\"?>\n")
buffer.WriteString("<rss version=\"2.0\" xmlns:atom=\"http://www.w3.org/2005/Atom\">\n")
buffer.WriteString("  <channel>\n")
buffer.WriteString("   <title>")
xml.EscapeText(&buffer, []byte(e.DeepDataMerge.LayoutConfig.SiteTitle))
buffer.WriteString("</title>\n")
/*
   so on
*/
buffer.WriteString("   <lastBuildDate>" + time.Now().Format(time.RFC1123Z) + "</lastBuildDate>\n")
buffer.WriteString("   <atom:link href=\"" + e.DeepDataMerge.LayoutConfig.BaseURL + "/feed.xml\" rel
```

Filtering and sorting posts by publication date

We don't want draft posts to show up!

```go
// slice it
var posts []parser.TemplateData
for _, templateData := range e.DeepDataMerge.Templates {
    if !templateData.Frontmatter.Draft {
        posts = append(posts, templateData)
    }
}

// sort by publication date
sort.Slice(posts, func(i, j int) bool {
    return posts[i].Date > posts[j].Date // assuming Date is Unix timestamp
})
```

Your RSS reader should pick it up, but let's sort them to ensure they make semantic sense.

Generating XML entries for the body

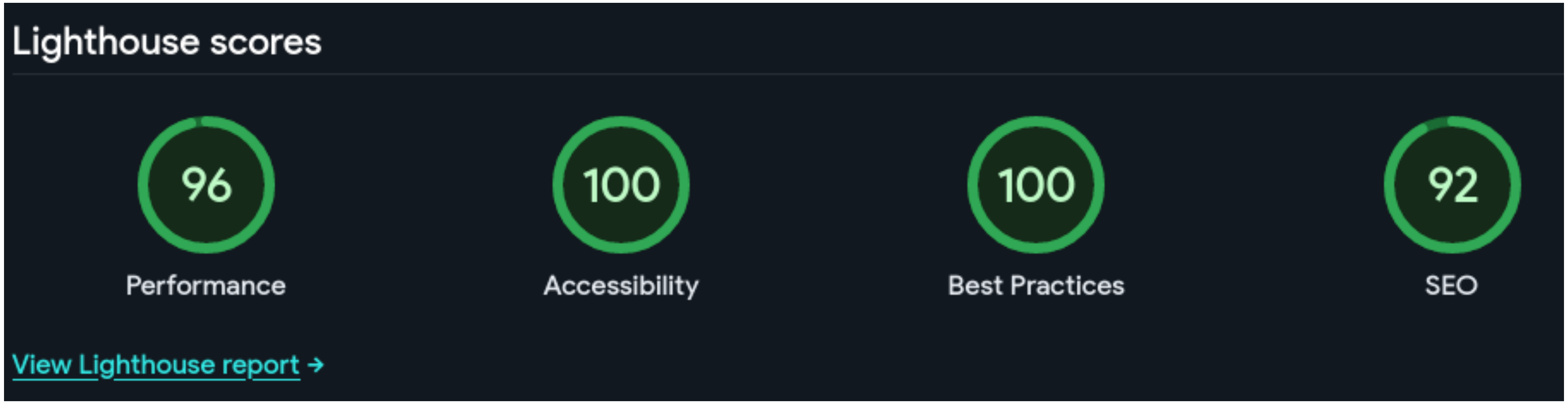
for each sorted post and writing to a file

```
for _, templateData := range posts {
    buffer.WriteString("    <item>\n")
    buffer.WriteString("      <title>")
    xml.EscapeText(&buffer, []byte(templateData.Frontmatter.Title))
    buffer.WriteString("</title>\n")
    buffer.WriteString("      <link>" + e.DeepDataMerge.LayoutConfig.BaseURL + "/" + string(template
    /*
        so on
    */
    buffer.WriteString("      <description>")
    xml.EscapeText(&buffer, []byte(templateData.Body))
    buffer.WriteString("</description>\n")
    buffer.WriteString("    </item>\n")
}
buffer.WriteString("  </channel>\n")
buffer.WriteString("</rss>\n")
outputFile, err := os.Create(e.SiteDataPath + "rendered/feed.xml")
```

For Robots 🤖

A sitemap tells web crawlers how your site is structured, outlining how often different pages are updated and helping search engines index your content efficiently.

```
User-agent: *
Allow: /

Sitemap: {{.BaseURL}}/sitemap.xml
```


Lighthouse scores

96 Performance

100 Accessibility

100 Best Practices

92 SEO

View Lighthouse report →

A buffer is used similar to the atom feed

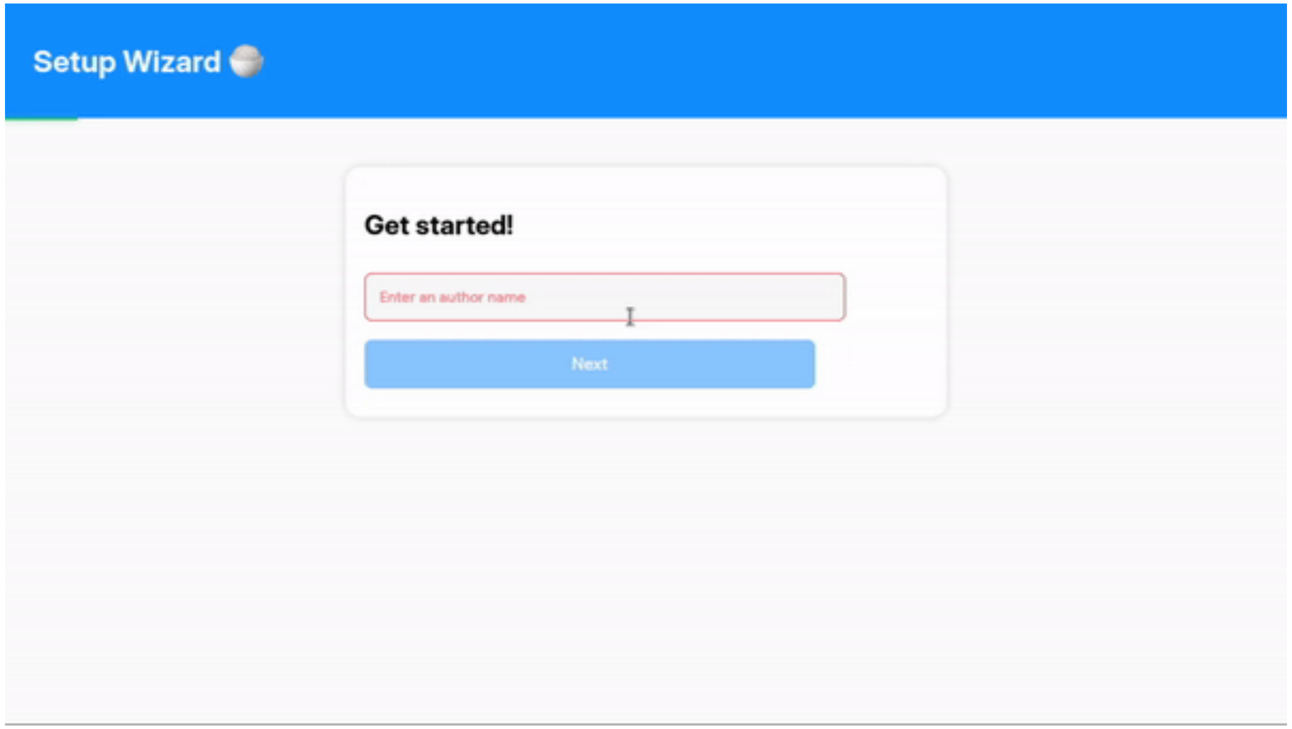
```
buffer.WriteString("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n")
buffer.WriteString("<urlset xmlns=\"http://www.sitemaps.org/schemas/sitemap/0.9\">\n")

// Sorting templates by key
keys := make([]string, 0, len(e.DeepDataMerge.Templates))
for k := range e.DeepDataMerge.Templates {
    keys = append(keys, string(k))
}
sort.Strings(keys)
tempTemplates := make(map[template.URL]parser.TemplateData)
for _, templateURL := range keys {
    tempTemplates[template.URL(templateURL)] = e.DeepDataMerge.Templates[template.URL(templateURL)]
}
```

```
for _, templateData := range e.DeepDataMerge.Templates {
    url := e.DeepDataMerge.LayoutConfig.BaseURL + "/" + string(templateData.CompleteURL)
    buffer.WriteString("\t<url>\n")
    buffer.WriteString("\t\t<loc>" + url + "</loc>\n")
    buffer.WriteString("\t\t<lastmod>" + templateData.Frontmatter.Date + "</lastmod>\n")
    buffer.WriteString("\t</url>\n")
}
buffer.WriteString("</urlset>\n")
```

For Humans 👾

An important ease-of-use feature was building a GUI

The wizard lets a user pick a theme, enter your name, pick navbar elements, and validates fields using **regex** checks so you don't need to worry about relative paths in baseURLs, canonical links, and sitemaps.



Bootstrapping themes

A catalog of themes are up on our GitHub, with a json index. Anna pulls the latest ones, populates a list of themes you can pick with the wizard.

```
func DownloadTheme(themeName string) error {
    zipURL := fmt.Sprintf("%s/%s/%s.zip", baseURL, tagVer, themeName)
    zipFile := fmt.Sprintf("%s.zip", themeName)

    fmt.Printf("Downloading %s...\n", zipURL)
    if err := downloadFile(zipURL, zipFile); err != nil {
        return fmt.Errorf("error downloading theme '%s': %v", themeName, err)
    }

    fmt.Println("Extracting files...")
    if err := unzip(zipFile, destDir); err != nil {
        return fmt.Errorf("error extracting files: %v", err)
    }

    fmt.Printf("Theme '%s' extracted to '%s' directory successfully.\n", themeName, destDir)
    if err := os.Remove(zipFile); err != nil {
        fmt.Printf("Warning: error deleting zip file '%s': %v\n", zipFile, err)
    }
}
```

Asynchronous

Signals are asynchronous processed using a channel (FormSubmittedCh)

```go
func (ws *WizardServer) handleSubmit(w http.ResponseWriter, r *http.Request) {

    // Decode JSON data from the request body into the LayoutConfig struct
    var config parser.LayoutConfig
    err := json.NewDecoder(r.Body).Decode(&config)
    if err != nil {
        ws.ErrorLogger.Println("Error decoding JSON:", err)
        http.Error(w, "Bad request", http.StatusBadRequest)
        return
    }

    err = ws.writeConfigToFile(config)
    if err != nil {
        ws.ErrorLogger.Println("Error writing config to file:", err)
        http.Error(w, "Internal server error", http.StatusInternalServerError)
        return
    }

    FormSubmittedCh <- struct{}{} // Signal that a form has been submitted successfully
}
```

After successfully completing the setup, the wizard launches a live preview of your site in a new tab

```go
// Called after successful form submission
err := DownloadTheme(config.ThemeURL)
if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    ws.ErrorLogger.Println("Error downloading and extracting theme:", err)
    return
}
```

This ensures robust error handling for various stages like JSON decoding, file writing, and theme downloading.

After successfully completing the setup, the wizard launches a live preview of your site in a new tab.

Deep Data Merge

- During the render process, each template has access to the complete data of the site via the DeepDataMerge struct
- This includes access to page template data, collections, configuration files, tags and a struct used to generate a JSON index of the site.

Deep Data Merge

```
type DeepDataMerge struct {
    // Templates stores the template data of all the pages of the site
    // Access the data for a particular page by using the relative path to the file as the key
    Templates map[template.URL]parser.TemplateData
    // Templates stores the template data of all tag sub-pages of the site
    Tags map[template.URL]parser.TemplateData
    // K-V pair storing all templates corresponding to a particular tag in the site
    TagsMap map[template.URL][]parser.TemplateData
    // Stores data parsed from layout/config.yml
    LayoutConfig parser.LayoutConfig
    // Templates stores the template data of all collection sub-pages of the site
    Collections map[template.URL]parser.TemplateData
    // K-V pair storing all templates corresponding to a particular collection in the site
    CollectionsMap map[template.URL][]parser.TemplateData
    // K-V pair storing the template layout name for a particular collection in the site
    CollectionsSubPageLayouts map[template.URL]string
    // Stores the index generated for search functionality
    JSONIndex map[template.URL]JSONIndexTemplate
}
```

Content Segregation

Content Segregation: Collections

- Anna lets you aggregate pages as collections and nested collections to filter content on various parameters
- Case in point: HSP Members Page (https://wonderful-pegasus-a6899b.netlify.app/collections/members/)
- Demo: Anna weekly progress (https://anna-docs.netlify.app/collections)

Content Segregation: Nested Collections

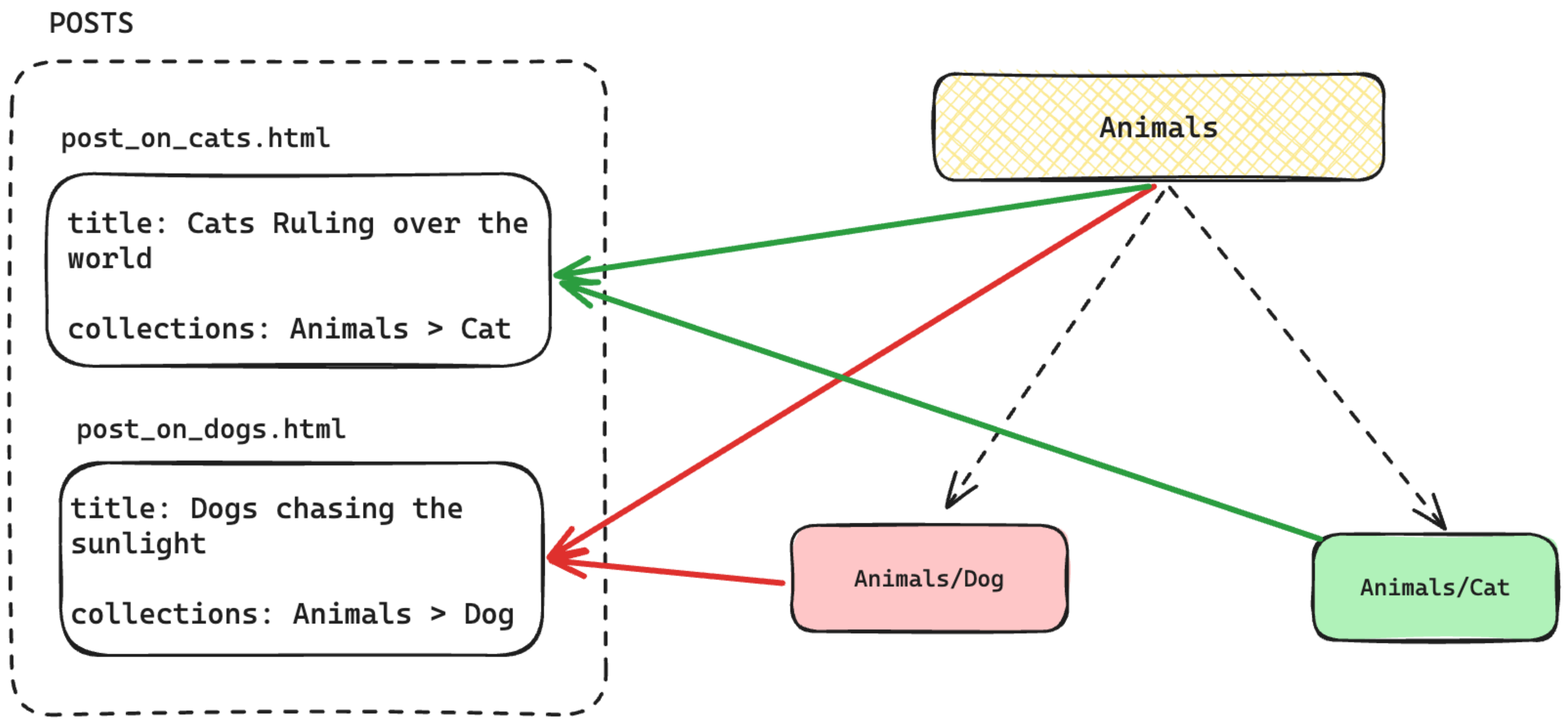
- A page can be added to a nested collection using the "parent-collection>child-collection" syntax
- This adds the page to both the parent as well as the child collection while maintaining hierarchy

```
func (p *Parser) collectionsParser(page TemplateData) {
    // Iterating over all sets of collections defined in the frontmatter
    for _, collectionSet := range page.Frontmatter.Collections {
        var collections []string
        // Collections will be nested using > as the separator - "posts>tech>Go"
        for _, item := range strings.Split(collectionSet, ">") {
            collections = append(collections, strings.TrimSpace(item))
        }
        for i := range len(collections) {
            collectionKey := "collections/"
            for j := range i + 1 {
                collectionKey += collections[j]
                if j != i {
                    collectionKey += "/"
                }
            }
```

Content Seggregation

```
            collectionKey += ".html"
            var found bool
            for _, map_page := range p.CollectionsMap[template.URL(collectionKey)] {
                if map_page.CompleteURL == page.CompleteURL {
                    found = true
                }
            }
            if !found {
                p.CollectionsMap[template.URL(collectionKey)] = append(p.CollectionsMap[template.URL
            }
        }
    }
}
```

Content Seggregation



POSTS
post_on_cats.html
title: Cats Ruling over the world
collections: Animals > Cat
post_on_dogs.html
title: Dogs chasing the sunlight
collections: Animals > Dog
Animals
Animals/Dog
Animals/Cat

Content Segregation: Tags

- Tags allow for further categorisation of content and has a similar implementation as collections renderer

Multi Site Render

- Multiple sites can be rendered independently by anna
- The render requirements can be configured via an "anna.json" file at the directory root

```
{
  "siteDataPaths": {
    "landing_page": "site/",
    "blog_page": "blogs/"
  }
}
```

Render Managers: Vanilla Render

- The muliple site rendering is made possible by RenderManager() functions which manage the directory paths on which VanillaRender() or LiveReload() must be invoked

```
func (cmd *Cmd) VanillaRenderManager() {
    // ...
    if cmd.RenderSpecificSite == "" {
        for _, path := range annaConfig.SiteDataPaths {
            // call vanilla render on all directories in anna.json
            cmd.VanillaRender(path)
        }
        // If no site has been rendered due to empty "anna.yml", render the default "site/" path
        if !siteRendered {
            cmd.VanillaRender("site/")
        }
    } else {
        siteRendered := false
        for _, sitePath := range annaConfig.SiteDataPaths {
            // call vanilla on passed directory render the path is valid
        }
    }
}
```

Render Managers: Live Reload

- The Live Reload Manager ensures that the site to be served, which is specified by the `-s` command-line flag, is a valid site path mentioned in `anna.json` and invokes LiveReload()

```
func (cmd *Cmd) LiveReloadManager() {
    // ...
    if cmd.ServeSpecificSite == "" {
        cmd.StartLiveReload("site/")
    } else {
        for _, sitePath := range annaConfig.SiteDataPaths {
            if strings.Compare(cmd.ServeSpecificSite, sitePath) == 0 {
                cmd.StartLiveReload(sitePath)
                return
            }
        }

        cmd.ErrorLogger.Fatal("Invalid site path to serve")

    }
}
```

Browser-side live reload

- Anna performs a re-render of the site on changes to the markdown files while serving
- It also performs browser-side live reload, eliminating the need for the user to reload the page manually
- An atomic boolean is used to determine if the page is to be reloaded or not

```go
func eventsHandler(w http.ResponseWriter, r *http.Request) {
    if !reloadPageBool.Load() {
        return
    }

    event := "event:\ndata:\n\n"
    _, err := w.Write([]byte(event))
    if err != nil {
        log.Fatal(err)
    }
    w.(http.Flusher).Flush()

    reloadPageBool.CompareAndSwap(true, false)
}
```

Browser-side live reload

- The server-sent event script is injected into the site pages during the live-reload mode to listen for changes and auto-reload the browser

```
func StartLiveReload() {
    // ...
    if lr.traverseDirectory(rootDir) {
        cmd.VanillaRender(lr.siteDataPath)
        reloadPageBool.CompareAndSwap(false, true)
    }
    // ...
}
```

```
{{ if $PageData.LiveReload }}
<script>
const eventSource = new EventSource("http://localhost:8000/events");
eventSource.onmessage = function (event) {
    location.reload();
};
</script>
{{ end }}
```

Future Additions

WebAssembly Integration:

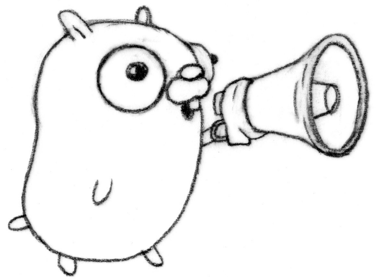
- Implement Wasm to enable native code compilation in the browser.
- Enhance static site functionality with dynamic elements and extended plugin capabilities.

Compatibility with Hugo SSG Themes:

- Provide Compatibility with Hugo themes, another widely used SSG implemented in GoLang.
- Access a wide range of themes from Hugo's extensive community ecosystem.

Usecases

We use anna personal websites, and wikis. An in the works, our redesigned college club website HSP (https://wonderful-pegasus-a6899b.netlify.app/)



hegde.live (https://hegde.live)

sudhir.live (https://sudhir.live)

polarhive.net/wiki (https://polarhive.net/wiki)

Thank you

Adhesh Athrey, Aditya Hegde, Anirudh Sudhir, Nathan Paul

6th July 2024

gh/anna-ssg