# Concept for Music Website

I need a special website **just for my music work.**

What are the **requirements?**

* Should load *really quickly* and be really *clean/minimal*
* Should have a theme that clearly shows that it’s about music *and* that shows the general genre/ambience/feel of that music.
* Should use a light Blog System/CMS as background, as it should support **static pages** (for each album) and **blog posts** (about the creation, things I’ve learned, progress, etc.)

What would be **nice to have**?

* A *very easy* way to access all my music => probably just a button to the LANDR page the distributor should give me
* A *very easy way* to find a specific genre of music, or specific project of mine => probably huge buttons at top/bottom
* Integration with a possible future webshop

These are **theme ideas**:

* I’ve made some cool ideas when I was researching the Rode Panda website. Has a custom font, style, etc.
* But I also want something *warmer*. Something with more icons, showing the guitar.
* And something more *professional*. Where you can easily find any music to listen and/or buy, with big buttons to do so.

These are **possible blogging systems**:

* **Hugo (seems to be best?)** => <https://gohugo.io/>
  + Install a BINARY on my computer itself
  + That is basically a local copy of the website, which I can edit as I like.
  + When satisfied, *export* that to a static website, and upload only that. (If possible, only upload the *changed* files.)
  + PRO: Easy explanation for how to create your own themes
  + CON: Not sure how easy it is to only “upload changed files”
  + Full guide on installing Hugo: <https://www.youtube.com/watch?v=c7vpcqA6SEQ>
* **Bludit** => <https://www.bludit.com/>
  + Really minimal, can be put directly on server, uses JSON files, flat files
  + Has a PRO version that would cost money … but don’t think I’ll ever need that?
* **Jekyll** => <https://jekyllrb.com/>
  + Use BINARY/COMMAND TOOLS on computer itself.
  + Then do the same as hugo. (Build website, then upload that.)
* **Ghost** => but that requires NodeJS, don’t think I can make that work on my shared server?
* **Worthy candidates:** October CMS, Hexo, Grav (already used), …

# How to use HUGO

**General:** At any time, when in your main folder, press **hugo server** to build the website and run it. You can now visit it at “localhost:1313”.

**URL:** <https://www.youtube.com/watch?v=c7vpcqA6SEQ>

**Step 1: Install PowerShell**. It’s just a free binary from GitHub.

**Step 2: Run it, as admin.**

* Just copy the command from the Chocolatey website to install that.
* Then copy the command from Gohugo’s website to install that.

**Step 3:** Go to where you want to create a new site, and type **hugo new site <name>**

**Step 4:** Go into that folder, into themes, and clone your desired theme into it.

**Step 5:** Configure it. Open **config.toml** in your favorite text editor.

* Most themes have a sample/default configuration, just copy that.
* Then edit to your liking.
* Mostly, this states directories, URLs, title, subtitle, creator credits, etc.

A good theme will use this configuration file *a lot* for the main properties, menus, header, site title and SEO stuff, etcetera.

**Step 6:** Add content.

* Type: **hugo new post/<name of post>.md** (Don’t forget the MarkDown extension!)

All your posts can be found in content/post in your directory. (This can be *different* between themes!)

**Step 7:** Each file will be created with **frontmatter**. At the top, between three dashes, is metadata/info about this specific post.

* Title = main title of post
* Data = date of publishing, also used for sorting/ordering
* Draft = true for unpublished, remove for published
* Image = to add a specific first image
  + Always add static content in the “static” folder => “blabla.png”
  + Of course, you can structure it however you want => create an uploads folder or something, then use the reference “uploads/blabla.png”
* Tags = [“sometag”, “some other tag”, …]
* Categories = [“some category”, …]

*No* need to put commas between, just newlines.

Here’s a link on YAML Frontmatter: <https://jekyllrb.com/docs/front-matter/>

**Step 8: Learn what all parts do:**

* Archetypes = how it creates every new post!
  + Basically, the *template* for the *frontmatter / starting content* for each new post.
* Data = ?? (you could override the default css or javascript of the theme)
* Content = holds your posts, pages, actual content
* Layouts = the files/code/structure used to render everything (so head.html, 404.html, etc.)
* Static = your static content (images, icons, scripts, etc.)
* Themes = all your themes (in your configuration, you should’ve picked one to activate)
* Config.toml = the hugely important config file

**Step 9: Do *not* edit the original theme.**

* Instead, move the files you want to edit into your main “layouts” folder, keeping the same structure as the theme uses.
* Then edit *that file* however you want.

**Step 10: Go loose.** Just use markdown to write your content.

# How to create a HUGO theme

**URL:** <https://www.youtube.com/watch?v=wcMqrb3v2SM>

**URL:** <https://retrolog.io/blog/creating-a-hugo-theme-from-scratch/>

**Step 0:** HUGO themes are just basic HTML files, but with special Go syntax peppered between.

Anytime you want to do something with Go, you enclose it in brackets: **{{ do something }}**

Do not forget to *close* blocks, such as loops and if statements, using **{{ end }}**

And usually, if you started a loop, you have a reference to the current looping object with **.**

For example: in a loop, I get the title of the current post with **.Title**

**Step 1: create the necessary stuff**

* Put a stylesheet (style.css) in the “static/css” folder.
* Fill the “head” with the common head stuff.
* Create a header, footer, and baseof template to stitch them together.
* Now fill index.html with something and you should at least see a website.

**Step 2: learn about blocks and partials**

You can always use “{{- partial "head.html" . -}}” to load a specific file from your *partials folder*.

Otherwise you can *define* a general *block* using “{{- block "main" . }}{{- end }}”

What does this do? In other files, we can create a block with this name, by typing {{ define "main" }}

Now it will be automatically included at that location in our *baseof.html* template.

**The idea is to use the DRY principle: Don’t Repeat Yourself.** Whenever you have functionality that is used on multiple pages, put it in its own file (a partial, most likely), and *call* it. Usually, that means code to *list all content*, *show one specific page*, *show metadata*, etc.

**Step 3:** **you determine the content structure, but it’s closely linked to your theme.**

Within the *content* folder, you can organize it however you like. However, this has *consequences* on how Hugo automatically interprets this.

Any top-level folder will be seen as a separate “section”. On my website, I have a folder “blog” and a folder “albums”.

The first one holds all my blog posts and news updates. The second one holds all my music.

When creating a new post, I simply do “hugo new blog/<name>.md”. When adding a new album, I do “hugo new albums/<name>.md”

Now comes the important part: **hugo automatically creates a tree from this, with two overview pages (blog and albums) at the top**. As such, if I were to just write a standard loop to display the last 5 pages, it would only display those two overview pages and nothing else.

Hugo calls these things **sections.**

So, how do we handle this?

**Step 4:** **learn how to use default Site and Page Params.**

This is how I display the latest 5 *blog posts* on the homepage.

“{{ range first 5 (where .Site.RegularPages "Section" "blog") }}”

* “range” indicates that we’re going to loop over the thing that comes after it
* “first 5” does what it says
* “where” is used to select a *subset*, much like the WHERE SQL clause.
* .Site.RegularPages means I want the actual pages, *not* those overview or section pages.
* “Section” “blog” means I only want those within the blog folder.

At first, I struggled with this. Why? Because I’d started Hugo with a standard theme and a *different* content structure. I tried to adapt the existing theme into my own, but it just wasn’t working out. (Whatever I tried, it either only found those top-level posts, or nothing at all.)

Only when I *completely changed the content structure*, and removed all code to write my own simple loop, did everything suddenly fall into place.

The lesson: **think about the best structure for your website before doing anything, and make the theme adapt to that properly.**

Once you’re inside this loop, though, everything is really straightforward:

* {{ .Title. }} gets you the current post title
* {{ .Summary }} gets you the auto post summary. (The length can be changed in the config.toml using the “summaryLength” param.)
* {{ .Data.Format “Jan 02 2006” }} gets you the date in that specific format. (You must always use that specific data when formatting. But the *order* in which you place elements, and *how* you write them, provide an easy way to format dates exactly as you want.)
* {{ .Permalink }} gives you the link to the article.

You can simply type these anywhere you like, and it prints the correct text to the page. So you can do <h2>{{ .Title }}</h2>, but also <a href=”{{ .Permalink }}”>Read more</a>, and it all just works.

**Step 5:** **learn to add your own Params.**

Each (markdown) file has so-called *frontmatter*. This is typed between three dashes and can be literally *anything*.

It will add a *title* and *date* by default. If you want to add a post to certain categories, you just do “categories: [a, b, …]”

But you can go even further. For example, my album pages automatically load themselves from Spotify, so you can play the album right from my website.

But … how does it know *which* album to load? That’s where the frontmatter comes in: in each album page, I define “album\_link: blabla” in the parameters.

In the code itself, I read that using .Params.album\_link.

**Important:** the *order* of operations/commands is a little weird to me. For example, in all programming languages (and real life) you do “if(section == albums) do … ”.

Here **equals** is “eq”.

And the order is … “{{ if eq .Section “albums” }}“ => the comparator/selector/whatever is pulled to the *front* and then wants its two arguments after it.

Also, **not equals** is “ne”.

**Step 6:** **learn about the different auto-generated pages.**

Hugo tries to do as much as possible for you, by doing the following:

First, by defining your *baseof.html* file, you create a wrapper around *all pages*. Usually, this contains your head, header, footer, and place to anchor main content.

Then …

* The homepage is displayed using *index.html*
* Anything that returns multiple results (an overview page for a section, all posts within a specific category, etc.) will use *list.html*
* Any single page will use *single.html*
* All of these pages will probably use several files from the *partials* folder. (Because they, as expected, contain only a part of a page, a module, one component.)

All the links and menus will automatically work. (For example, if you use the Permalink on a Section page, it will automatically load a list. But the Permalink on an individual page, will automatically go that page, using the single template.)

If you want to differentiate even further, you need to use the Go syntax to determine which page you’re on

**Bonus: some markdown realizations**

* To force a newline in markdown, end a line with two spaces (or more)
  + This is useful for when you need a newline, but don’t want it to start a new paragraph
  + Without this, though, whitespace collapses and lines underneath each other will just render after each other.
* To add tab space between stuff, use the HTML entity &emsp;
* Use <h1> for displaying the *title* of a page. Then use <h2> (## in MarkDown) for all headings *within the actual post*.
  + This makes more (semantic) sense to search engines and browsers.
  + If you don’t do this, for example, and make all headings <h1>, the screen reader function in Firefox will not work.
* Sometimes … I accidentally added the wrong paragraph/line break character?
  + In some versions of Chrome, this would show up as a black box with a white cross in it.
  + This character is Unicode 2029
  + So to solve this, simply do a “find in files => regular expression => \x{2029}” and replace all of that with “” (empty space)

**Lesson #1:**