HBML Tutorial

How to use HBML from simple markdown to complex macros

Introduction

Welcome to the HBML book! In this book we cover the entirety of HBML from simple files to complex macros and external dependencies

But to start, we need to introduce some useful terms we use:

- **Void elements** are HBML elements that can't have anything in it. For example, the `meta` tag is a void element
- The **arrow operator** is the > character. This is used after a tag, which can be empty
- A **child element** is any element that's nested inside another element called the **parent element** . For example in `div { h1 > "Example" }`, the `div` is the parent element and the `h1` is the child element in respect to each other
- **Implicit tags** are tags that are defined implicitly. For example `{ "Hello World!" }` is actually `div { "Hello World!" }` just that the `div` tag is implicit

Getting started

First off, you'll need to have HBML installed:

```
npm install hbml -g
```

To check this, you can run `hbml -h` and you should get the HBML help message

What we'll cover

In this chapter we'll cover:

- Making your first HBML file
- Setting up a project
- Building and linting HBML
- Including dependencies and local files

Your first HBML file

To start off, let's make a file called `hello_world.hbml`. In this file we'll start off with a default template

```
:root {
    head {
        title > "Hello from HBML!"
    }
    body {
        /* Your first HBML file */
        h1 > "Hello from a HBML file!"
    }
}
```

But what does this mean?

You may notice that the structure is pretty similar to a simple HTML file which is intentional. HBML is built on HTML with brackets not tags. In this sense, writing in HBML is functionally no different to writing in HTML, but hopefully a nicer experience.

But what are the > characters? If you have something with one element inside it, you can use the arrow operator. This is the same as putting the element after the arrow in brackets, but easier to read.

You'll also have noticed the comment line just before the `h1`. Comments in HBML are indicated by a starting `/*` and closing `*/` which can be on different lines. We discuss comments more when we talk about building HBML.

IDs

An important part of HTML is tag IDs. In HBML we use the `#` character to indicate an ID. For example `h1#mainHeader` would be the same as `<h1 id="mainHeader">` in HTML.

IDs are placed after a tag type and are optional

Classes

Classes are also a very important part of any webpage. In the style of CSS selectors, to specify a class on a tag we use the `.` character. For example `div.header` would be equivalent to `<div class="header">` in HTML.

Classes are placed after IDs and are optional

Attributes

Tag attributes are placed inside square brackets and use the standard style. For example `meta[charset="UTF-8" someOtherAttribute]` would be the same as `<meta charset="UTF-8" someOtherAttribute/>` in HTML.

Unique attributes

Any attribute can be a *unique attribute*. This means that if duplicates of the attribute are found, then the last one is used. If a duplicated attribute is not unique, then the values are placed together with a space separating them. For example, `lang` is a unique attribute so `html[lang="en" lang="de"]` would turn into `<html lang="de">`; but `class` is not a unique attribute so `p.lead[class="bold"]` or `p[class="bold" class="lead"]` would both become `<p class="lead bold">`.

You can override the default unique attributes by adding or removing attributes like this:

```
/* Make the 'class' attribute unique */
:unique +class
/* Make the 'lang' attribute not unique */
:unique -lang
```

Attributes are placed after classes and are optional.

Implicit tags

One of the large strengths of HBML is it's implicit tags. As the name suggests, these are implied tags and allow your code to look cleaner and be more readable.

Implicit tags are either `div`s or `span`s depending on the parent element type and default to `div`s. To give you an idea of how to use implicit tags, all of the following result in an implicit tag as the root tag:

- `> "This text is under an implicit tag"`

  Nothing before a curly bracket or arrow will create an implicit tag
- `#tagID { "An implicit tag with an ID" }`

  Tag types are optional, but that doesn't mean you can't put an ID, classes, or attributes
- `.bold[href='some_page.html']`

Strings

Strings in HBML can use one of three delimiters `'`, `"`, or `` ` ``. If you use the `'` or `"` delimiter, newlines won't be included in the resulting string, but using `` ` `` will include newlines.

Strings are also put through a substitution process to turn characters that might cause problems in HTML into their HTML codes such as < being replaced with `&lt`

Projects

Very rarely will you be working with single HBML files in isolation. It's much more common for you to be using them in a project with multiple HBML files.

Because of this, you can use HBML alongside a project just like normal.

To do this, run `hbml init` in the project directory. This will create a `hbml.json` file with the default value below

```
{
  "build": {
    "src": ["."],
    "output": "html"
  },
  "lint": {
    "src": ["."],
    "output": "html"
  }
}
```

Macros

HBML has allows you to simplify your code with macros. Macros are scoped and mutable and have standard defaults available to use. In this chapter, we'll introduce the basics of macros and how to use them, then we'll delve into more complex macros with conditionals that really give macros their power.

Simple macros

The simplest type of macro are replacement macros. These don't consume any elements, and are very simple to define.

We always define macros the same way

```
--macro_name {
    // What the macro expands into
}
```

You can also use the arrow operator if it expands into a single element such as if your macro will expand into a string or table.

Calling macros

To call a macro, all you need to do is type a colon (`:`) then the name of the macro you want to call. For example, in all HBML files, you'll see the `:root` macro which expands into required HTML attributes.

Examples

Page text in macros

One use for simple macros like this is putting the text of a page into macros to make writing the page a bit cleaner. To do this you might have something like the following

```
--main_header > "Place your page header here"
--main_body {
    p > "Some text here"
    img[src="photo.png"]
}
```

Head tags

Large projects often require using lots of external scripts (FontAwesome, JS libraries, font scripts, etc.). And for most of the files you use, there will be duplicated sections in the head section of your page. So, you could define a macro that includes all the repeated head sections and then call that in your pages

```
--head-base {
    script[src="script1.js"]
    link[rel='stylesheet' href='style.css']
```

```
}
```

Macro children

Macros can do more than just turn into text. If macros have child elements, you can make use of them in the macro.

When you define a macro, you have access to the `:child` and `:children` macros in the definition. The `:child` macro will get the next child element under the macro call or, if no child elements are left, an empty string. The `:children` macro will get all the remaining elements under the macro, or an empty string of there aren't anymore.

The best way to show this is to see some examples, so that's what we've done

Examples

A list macro

This example is a bit pointless considering how simple it is to define a list, but there we are

```
--list {
    ul {
        li > :child
        li > :child
        li > :child
    }
}
```

Elements in a block

If you need everything on your page to be nested inside several items to make sure it's formatted correctly, you might use a macro for that

```
--block > .class1 > .class2 > .class3 > :children
```

Notice how you don't have to surround `:children` in brackets. Because it looks like one element, we treat `:children` as one element when reading macro definitions. When it gets expanded, all the child elements will be grouped under `.class3` as you would expect.

An important note

When using `:child`, it may be the case that some child elements remain un-used. If and when this

happens, a warning or error is raised about it depending on your build set-up.

Conditionals

In the previous section, we gave an example of a simple list macro. However, that would only work for three child elements. In this section, we'll introduce the `:consume` and `:consume-all` macros that make macros much more powerful.

The `:consume` macro checks how many child elements are required for the section under it, and will only return that if there are the right number of children left.

For example, using our list example again, we might want to make sure that for each `:child`, there actually is one. So we could have

```
--list > ul {
    :consume > li > :child
    :consume > li > :child
    :consume > li > :child
}
```

This will mean that if we only gave two elements, the list would only have two elements in it now because the last `:consume` would see that it required one child, and that none were left, so it wouldn't expand into anything.

But, we can do better. What if we gave four elements. The list would still only have three elements, and we'd have one element left un-used. This is where `:consume-all` comes in. `:consume-all` acts just like `:consume` but will repeat until there aren't enough child elements left.

This means we can make our list macro much better

```
--better-list > ul > :consume-all > li > :child
```

The `:unwrap` macro

The last macro is the `:unwrap` macro. This macro takes some elements, and "unwraps" them. Comments and strings are left alone, but elements have their children returned.

```
:unwrap {
    { "Text" }
    "More text"
    .class { p > "A paragraph" }
}
```

```
// Equivalent
"Text"
"More text"
p > "A paragraph"
```

Examples

A two columned table

When writing documentation, you may find yourself making a lot of tables with two columns in them.

So, you might define a macro similar to this one

```
--2table > table {
    thead { tr { td > :child td > :child } }
    tbody > :consume-all {
        tr { td > :child td > :child }
    }
}
```

If we then consider equivalents, we see what happens if `:consume-all` doesn't use every element

```
// macro
:2table { "Heading 1" "Heading 2" "Row 1" "Value 1" }

// equivalent
table {
    thead { tr { td > "Heading 1" td > "Heading 2" } }
    tbody > tr { td > "Row 1" td > "Value 1" }
}
```

When we put the right number of elements under the macro call, it works as expected. But, if we add one more element under our macro

```
// macro
:2table { "Heading 1" "Heading 2" "Row 1" "Value 1", "Row 2" }

// equivalent
table {
    thead { tr { td > "Heading 1" td > "Heading 2" } }
```

```
    tbody > tr { td > "Row 1" td > "Value 1" }
}
```

We see that the new value is ignored because the `:consume-all` section in the macro required two elements but only got 1. If we added an extra value, the macro would add in a new row to the table

An interesting table macro

If we wanted to make our table macro nice and generic, we might make something like this

```
--table-row > tr > :consume-all > td > :child
--table > table > :consume-all > :table-row > :unwrap > :child
```

This might look a little daunting, but we can work through it; especially if we specify what kind of inputs the macro expects.

Let's start with `:table-row`. This macro expects to be given a load of elements, each of which will be put into a `td` rag and all of them under a `tr` tag.

Now, `:table`. This expects to be given several elements each of which with child elements (see the example usage below). For each given element, it'll expand them (returning the elements children), and pass that to `:table-row` to generate a table row. This is then repeated for each given element, and then all of it gets wrapped up in a `table`.

```
:table {
    { "Heading 1" "Heading 2" }
    { "Row 1" "Value 1" }
    { "Row 2" "Value 2" }
}

// Equivalent
table {
    tr { td > "Heading 1" td > "Heading 2" }
    tr { td > "Row 1" td > "Value 1" }
    tr { td > "Row 2" td > "Value 2" }
}
```

Scopes

Scopes are important in discussing macros and understanding how they work can be very useful.

A scope is a section of HBML that has access to the same macros and is either the root scope or under an element.

For example, if we had

```
--macro1 > "Hello"
{
    --macro2 > "World"
}
```

then we have two scopes; the root scope, and the scope inside the brackets. Because the macro `:macro2` is defined in the brackets, it's not usable outside of them because it "doesn't exist".

Standard macros

This section is a for checking what macros you have by default. These are the standard library macros.

The headings below tell you when you have access to the macros.

All the time

In macro definitions

Simple import files

HBML allows you to separate your projects into multiple files and import them later. We steal more syntax from CSS and use define importing files like this

```
@import path/to/file
```

The path you give can optionally end in `.hbml`. If it does not, the builder will add it automatically for you. The path can also be a URL if you want to use an external macro library.

When you import a file, you only import the macros in the root of the file. These macros behave the same as if they were defined in the file you import them into. So, if you import a file with the macro `:example` in, you can use `:example` in your file just the same.

Collisions

When you import macros, the builder will check for collisions, meaning macros with the same name. If it finds any, it'll give you an error. Otherwise, it adds the macros into the current scope. You can, however, redefine macros manually without raising an error. This has been done for two reasons:

1. When importing a file you're not immediately presented with every macro in it and you might accidentally overwrite a macro you needed by importing one with the same name. If you intend to do this, you can easily get rid of the error with namespaces
2. Macro importing can sometimes include useful macros that work most of the time but will occasionally need changing. We recommend this change be done manually to avoid an error and that you can use the macros the same without the need for namespaces

Namespaces

When importing a file, you can optionally specify a namespace for all the macros in that file. If you do, any macro used from that file will need to be prefaced with the namespace of that file.

For example, if `a.hbml` contains `:example`, you could import it as `@import a namespace-a` and then use `:example` from `a.hbml` as `:namespace-a:example`.

CLI Commands

This chapter contains the different CLI commands and how to use them. If you prefer the CLI help info, you can always run `hbml <command> -h` for help on a command

Build

The `hbml build` command will build files or a project into HTML.

This command can either be run as `hbml build project` to build your project (see the project section for more info on projects); or with some number of files or directories you want to build.

When running in non-project mode, you can specify various flags to alter the build process. These flags are ignored in project mode.

Lint

The `hbml lint` command will lint files or a project.

This command can either be run as `hbml lint project` to build your project (see the project section for more info on projects); or with some number of files or directories you want to build.

When running in non-project mode, you can specify various flags to alter the build process. These flags are ignored in project mode.

Init

The `hbml init` command will initialise a directory (by default the current one) with the standard HBML project structure.

Reverse

The `hbml reverse` command allows you to convert HTML back into HBML.

The command accepts a list of paths, an optional output prefix, and a verbose output option.

It uses a HTML parser to tokenise the input which is then converted to HBML tokens and turned into linted HBML with the default linting options with one change to make elements with one child prefer the arrow operator over using brackets.