BEGINNER GUIDE

INTRODUCTION

Welcome to the Beginner Guide for DocPad. By the end, you'll have all the knowledge necessary to create amazing powerful websites and applications with DocPad. You won't need any special training, and if you are a web developer then you'll already feel right at home.

For this guide we'll create a blog and explain step-by-step how it's done. A blog is just a listing of posts/articles as well as some pages. Posts are intended to allow regular new content, like status updates, to be published, whereas Pages are more permanent, usually providing general background information.

Great, let's get started! Oh, and if you ever get stuck, you have a question or need help, then jump onto any of our official support channels and we'll be right with you. Cheers!

Click the headings to proceed to the particular section.

CREATING THE STANDARD PROJECT STRUCTURE

Before we get hacking away on content, we have to create our project and set it up with DocPad's standard structure. To do this, we'll run the following:

```
mkdir my-new-websitecd my-new-websitedocpad run
```

When prompted for a skeleton, select `No Skeleton`, which will set up our standard directory structure automatically. The `docpad run` command will also keep running once the action has completed successfully. This is because it also watches for changes and will regenerate our website when they occur

website as we go.

For now, we'll keep `docpad run` running but, whenever you want to stop it, just hit `CTRL+C` on your keyboard to exit it.

ADDING THE HOME PAGE

Let's create our first document; the *Homepage* for our website. Create the document `src/render/index.html` and give it the following content:

```
<html><head>    <title>Welcome! | My Website</title></head><body>
 <h1>Welcome!</h1>    <p>Welcome to My Website!</p></body></html>
```

Once you've saved it, open up http://localhost:9778 (or refresh if you already had it open) and you'll notice that DocPad has already regenerated your website and see the page we created rendered inside the browser. Fantastic!

ADDING THE ABOUT PAGE AND A LAYOUT

Now that's done, we'll want to add our "About Me" page so that people browsing our website will know who we are. To do this, let's create a new document at `src/render/about.html` and give it the content:

```
<html><head>    <title>About Me | My Website</title></head><body>
 <h1>About Me</h1>    <p>I like long walks on the beach. <strong>Plu
s I rock at DocPad!</strong></p></body></html>
```

Now if we go to http://localhost:9778/about.html, we'll be able to see that page in our browser. Awesome!

However, duplicating that layout information inside our *Homepage* and our*About* page is pretty redundant. For instance, if we wanted to change the contents of `<head>` to something else, then we'll have to change it in two

Layouts wrap around our documents, so we can define the surrounding areas of a document only once. Let's move the wrapper stuff of our two documents into a new layout located at `src/layouts/default.html.eco`, so that we end up with the following:

- `src/layouts/default.html.eco`

```
<html><head>    <title><%= @document.title %> | My Website</title></head><body>    <h1><%= @document.title %></h1>    <%- @content %></body></html>
```

- `src/render/index.html`

```
---title: "Welcome!"layout: "default"isPage: true---<p>Welcome to My Website!</p>
```

- `src/render/about.html`

```
---title: "About Me"layout: "default"isPage: true---<p>I like long walks on the beach. <strong>Plus I rock at DocPad!</strong></p>
```

However, if you go to either the *Homepage* or the *About* page on our web server, you'll notice that their content is just the layout, and don't actually contain any of the document content. This is because we haven't installed the templating engine for our layout yet.

We've also added a new `isPage` attribute, this is our own custom attribute and has no special meaning in DocPad. However, we will give it special meaning later on when we add our menu listings :)

INSTALLING THE TEMPLATING ENGINE

Templating Engines allow us to embed abstractions inside our documents - which is why we have to use them for layouts, as a layout is an abstraction, an abstraction that wraps around a document and outputs the document's content in a specific way :)

us via the eco plugin. So let's install that now by quitting DocPad (`CTRL+C`), and running:

```
docpad install eco
```

Once installed, if we run DocPad again (`docpad run`) and then go our*Homepage* or *About* page, we'll see that the content actually contains the document content. In other words, it has rendered correctly through our eco templating engine. Woot woot!

Now, why did we use `<%=` for the document's title, but `<%-` for the document's content? The reason for this is that `<%=` will escape the data we give it before outputting the data, which allows special characters to be interpretted as normal characters. On the other hand, `<%-` will output without performing any escaping, leaving special characters intact. So, why should we escape some things but not others? Generally, whenever the data we want to output is HTML (e.g., the document's content) we want to output it "as is", without any escaping. On the other hand, if we didn't escape document titles, titles such as `3 is > 2` would end up as `<title>3 is > 2</title>`, which is invalid HTML due to the superfluous closing element character, `>`. Thus, using escaping would result in `<title>3 is > 2</title>`, which uses the HTML entity code to represent the special character. This is valid HTML and looks exactly the same to the user :)

A NOTE ON PLUGINS

Even though we have chosen to use the Eco templating engine in this guide, there are plenty of others we could use too! We find Eco is very beginner-friendly, since it's the most like HTML. However, after experimenting with some of the others, you may find you'd like to use them instead. Adding support for a new templating engine, pre-processor, or whatever is as simple as installing the plugin for it.

DocPad also supports more than just templating engines, though. We have a whole range of different plugins to do all sorts of things, so be sure to check them out!

You can find the full listing of plugins we have on the Plugins page.

A NOTE ON RENDERING

Now, the reason we can support multiple templating engines is because of the way DocPad renders things, file extension by extension. Thus, by creating the file `default.html.eco`, we can render from `eco` to `html`. Conversely, if we tried `default.eco.html`, we'd be trying to render from `html` to `eco`, which wouldn't actually be very useful or even possible. However, other combinations can be rendered in both directions, such as CoffeeScript to JavaScript and JavaScript to CoffeeScript, provided you have the right plugins installed. You can also mix and match extensions, such as `default.html.md.eco`, which renders from `eco` to `md` (markdown) and then `md` to `html`. This allows you to apply abstractions with eco to your markdown documents. Awesome! For now, though, it's way out of our scope, so let's get back on track!

A NOTE ON META DATA

What on earth was with the stuff between the `---` at the top of our documents? That stuff is our *meta data*. It is where we can define extra information about our document, such as its title, layout, date, whatever. You are not limited to what you can define here; it's up to you. However, there are some pre-defined properties that serve special purposes (such as layout, which we just used).

You can learn more about meta data and all the special properties on ourMeta Data page.

ADDING THE LIVE RELOAD PLUGIN AND BLOCKS

ADDING THE LIVE RELOAD PLUGIN

Sweet, so we're doing well so far! We've got two documents, and a layout to abstract them. As we are making changes, it sure would be nice if our browser refreshed the page automatically! We can do this with the Live Reload Plugin, so let's install that now:

```
docpad install livereload
```

Once you've restarted DocPad, whenever you make a change to the files, you'll notice the browser still doesn't refresh automatically to show the changes! Why? The reason is that we have to add our *Blocks*.

ADDING THE BLOCKS

Blocks are a way for plugins and even ourselves to be able to easily add scripts, styles and meta elements to our pages. In the instance of the Live Reload Plugin, it needs the *Script Block* to be able to inject the needed scripts into the page that refresh the browser.

So to add the three default blocks to our layout, we'll update our `default.html.eco` layout to become:

- `src/layouts/default.html.eco`

```
<html><head>    <title><%= @document.title %> | My Website</title>
  <%- @getBlock("meta").toHTML() %>    <%- @getBlock("styles").toHTM
L() %></head><body>    <h1><%= @document.title %></h1>    <%- @conte
nt %>    <%- @getBlock("scripts").toHTML() %></body></html>
```

Saving that, and manually reloading our browser, we'll notice that our page now has the necessary scripts for the Live Reloaded Plugin automatically injected right where the scripts block has been output. Now if we make a change to any of the files, we'll notice the browser will automatically refresh. Amazing!

In the next part, we'll work with blocks some more by adding our assets to them.

ADDING ASSETS

It's time to start adding some assets. Before proceeding with this section, please read the DocPad Overview Page so you know what each of the directories inside our website structure are for.

IMAGES

Let's add our logo to our layout's header. We'll download the DocPad logoand place it in our `static` directory at `src/static/images/logo.gif` (binary files should *always* go in the `static` directory). Then, we'll add it to the body of our layout, to show our logo on each page:

- `src/layouts/default.html.eco`

```
<body>  <img src="/images/logo.gif" />  <h1><%= @document.title %></
h1>  <%- @content %>  <%- @getBlock("scripts").toHTML() %></body>
```

If you are downloading the file directory into the specified location, you may have to restart DocPad. We're working on this.

STYLESHEETS

Now let's make all of our `h1` headers red, by adding a stylesheet file in our render directory at `src/render/styles/style.css` that contains:

```
h1 {    color: red;}
```

Then, to include it in our pages, we'll update the *styles Block* in our `default.html.eco` layout to:

```
<%- @getBlock("styles").add(["/styles/style.css"]).toHTML() %>
```

Upon saving, we'll notice that our browser will automatically reload, and that our CSS file will be injected into the layout making our header red!

SCRIPTS

Now let's add a nifty loading effect using JavaScript and the jQuery JavaScript Library. As always, there's plenty of other JavaScript Libraries you

To do this, we'll first download the jQuery library file and put it in our `static` directory at `src/static/vendor/jquery.js`.

The reason we use the `static` directory for vendor files is that it is extremely unlikely we'll ever want to render any vendor files, so having them there is a good choice for consistency and speed. Whereas, we will probably eventually want to render our own scripts and styles with something, so generally we'll just put them in the render directory to make the transition to rendering engines easier.

Now that we have included jQuery in our project, we'll add our nifty loading effect by adding a script file at `src/render/scripts/script.js` that contains:

```
(function(){    $("body").hide().fadeIn(1000);})();
```

Now that's done, let's add those files to our *scripts Block* in our `default.html.eco` layout:

```
<%- @getBlock("scripts").add(["/vendor/jquery.js","/scripts/script.j
s"]).toHTML() %>
```

Upon saving, we'll notice that our content will fade in over a duration of two seconds. Nifty!

Now, some of you may wonder why we omitted the jQuery onDomReadyhandler in our script file. While off-topic for DocPad, the reasoning for this is that only code that requires the entire DOM to be loaded needs it. For instance, if your script requires a DOM element that is positioned after our `script` tag, then it would be useful.

In this instance, as we inject our scripts into our `body` element, we already have access to the `body` element, and therefore can start our `fadeIn` animation immediately. This avoids the page loading, then, the DOM loading after a delay, followed by our fade-in with an undesirable "popping" effect.

GETTING THE BENEFITS OF PRE-PROCESSORS

Pre-Processors are amazing things. They allow us to write documents in one language (the source language), and export them to a different language (the target language). This is extremely beneficial, as it allows you to use the syntax that you enjoy, instead of the syntax that you are sometimes forced to work with. Most importantly, however, pre-processors often offer more robust and clean functionality than the target language supports out of the box, allowing you to make use of modern developers while still working with old languages.

USING MARKDOWN, AN HTML PRE-PROCESSOR

HTML's verbose syntax is terrible for writing content that is more text than markup (e.g., articles, comments, etc.). Fortunately, Markdown (one of the many HTML Pre-Processors available to us as Plugins) comes to the rescue!

Install the Marked Markdown Plugin by running `docpad install marked` .

Then, rename the *About* page we created earlier from (`render/about.html`) to (`render/about.html.md`), to indicate that we want to render from Markdown to HTML, and open it. Writing in Markdown, update its content (leave the existing meta data section as it is) to become:

```
I like long walks on the beach. **Plus I rock at DocPad!**
```

Which gives us the same result, but with all the benefits of Markdown!

This simple procedure can be followed irrespective of your desired HTML pre-processor.

Sweet, you're now ready to party, Markdown-style! ;)

USING STYLUS, A CSS PRE-PROCESSOR

Open the Stylesheet document we created earlier (`render/styles/style.css`). CSS really hasn't come that far over the years and, thus, it has absolutely no abstractions available to us, making it incredibly verbose and painful to write. Fortunately, Stylus (one of the many CSS Pre-Processors available to us) is our saviour!

Install the Stylus Plugin by running `docpad install stylus`.

Then, rename `src/render/styles/style.css` to `src/render/styles/style.css.styl`, to indicate we want to render from Stylus to CSS, and open it. The reason why we created the style file in `render` and not in `static` is now obvious: if the Stylus stylesheet file were in `static/styles/` folder, it would not have been pre-processed before copying to `out`.

Using Stylus syntax, update the stylesheet's content to become:

```
h1    color: red
```

Which gives us the same result as before, but with all the benefits of Stylus.

This simple procedure can be followed irrespective of your desired CSS pre-processor.

Sweet, you're now ready to rock the house with Stylus!

USING COFFEESCRIPT, A JAVASCRIPT PRE-PROCESSOR

Sometimes people can get quite irritated with JavaScript's verbosity, and very annoyed at its nit-picking, such as when they forget a single comma somewhere and their entire app breaks. Fortunately, CoffeeScript (one of the many JavaScript Pre-Processors available to us) restores our sanity!

Install the CoffeeScript Plugin by running `docpad install coffeescript`.

Then rename `render/scripts/script.js` to `render/scripts/script.js.coffee`, to indicate we want to render from CoffeeScript to JavaScript, and open it.

Using CoffeeScript, we can update our file's content to become:

```
$("body").hide().fadeIn(1000)
```

Which gives us the same result, but with all the benefits of CoffeeScript.

This simple procedure can be followed irrespective of your desired JavaScript pre-processor.

Sweet! Now you're ready to relax, with a rich cup of CoffeeScript.

ADDING SOME TEMPLATE DATA AND TEMPLATE HELPERS VIA A CONFIGURATION FILE

PURPOSE OF A CONFIGURATION FILE

The DocPad Configuration File allows us to configure our DocPad instance, listen to events and perform some nifty abstractions.

Consider the case where our document title is empty. With our current solution, the title of the page would be `| My Website`. A page title of `My Website` would look far better when our document doesn't have a title.

To handle this, we update our title code in our `default.html.eco` layout template to become:

```
<title><%= if @document.title then "#{@document.title} | My Website"
 else "My Website" %></title>
```

Which would achieve the immediate goal, but then would mean that we would have to update the website title in two places if we want to use anything other than `My Website`. Considering this a common requirement, it would be nice if we could abstract it out, say, into a configuration file!

DocPad should have created an application-wide configuration file for us at `/docpad.coffee` (in the project root) when we initialized the project. If not, let's create it, with the following contents:

```
# Define the ConfigurationdocpadConfig = {    # ...}# Export the Con
figurationmodule.exports = docpadConfig
```

Notice that the `docpadConfig` object is written in CoffeeScript's version of JSON.

You'll have to restart DocPad so that DocPad can become aware of the configuration file. From then on, DocPad will automatically reload your configuration when changes occur.

The first part of this configuration is where we actually define our configuration (where the `# ...` is located), and the second part is a Node convention for exporting data from one file to another. Whenever we add some configuration, you'll want to add it to the `docpadConfig` object we just defined.

For more information on configuration files and what configuration is available to your, refer to our Configuration Page.

USING TEMPLATEDATA FOR ABSTRACTIONS

Everything that is available to our templates is called TemplateData. For instance, `@document` is part of our template data. To be able to abstract out something that our templates will use, we will need to extend our template data. We can do this by modifying our template data configuration property in `/docpad.coffee` like so:

```
docpadConfig = {    templateData:        site:            title: "My
 Website"}
```

With that, our website title is now abstracted and we can update our title element in the `default.html.eco` template:

```
<title><%= if @document.title then "#{@document.title} | #{@site.tit
le}" else @site.title %></title>
```

However, if we really wanted to (and we probably do) we can abstract out that logic into a function inside our template data.

ABSTRACTING LOGIC INTO TEMPLATE

HELPERS

When using `.coffee` or `.js` files to define our Configuration File, we are allowed to define functions. Doing so allows us to use functions within our template data, which we call *Template Helpers*.

When calling a template helper, the scope of the template helper is exactly the same as the scope of whatever is calling it. This makes abstracting out logic really easy. Let's see what our object in `/docpad.coffee` would look like:

```
docpadConfig = {    templateData:        site:            title: "My
 Website"        getPreparedTitle: -> if @document.title then "#{@do
cument.title} | #{@site.title}" else @site.title}
```

And the title of our layout template, `default.html.eco`, would become:

```
<title><%= @getPreparedTitle() %></title>
```

Now that is awesome! While this was a simple example, we can use it to do some really cool stuff. For instance, here is an example of it being used to localize dates into French.

If you're writing a plugin, you can use the `extendTemplateData` event to extend the template data.

ADDING A MENU LISTING FOR OUR PAGES

Remember our *About* page? Wouldn't it be nice if, when we list more pages, our menu updates automatically? It sure would, so let's do that!

UPDATING OUR LAYOUT

Open your default layout, and add the following before the `h1`:

```
<ul>    <% for page in @getCollection("html").findAll({isPage:true})
.toJSON(): %>        <li class="<%= if page.id is @document.id then
'active' else 'inactive' %>">            <a href="<%= page.url %>">
              <%= page.title %>            </a>        </li>    <%
end %></ul>
```

Save it, and BANG! Now we've got our navigation menu on each page! Wicked. So what does that do? Well first it uses the `getCollection` template helper to fetch the `html` collection, which is a pre-defined collection by DocPad that contains all the HTML documents in our website. Then, with that collection, we find everything that has a `isPage` attribute set to `true`. (We defined it earlier, when first applying our layout to our pages.) Then, we convert the result from a Backbone Collection / QueryEngineinto a standard JavaScript Array using `toJSON`.

That's a bit of a mouthful, but give it a while and you'll be a pro in no time.

There is one major inefficiency with the above approach. Can you guess what it is?

Performing the query every single time we render a layout is a bit silly, as the results won't change each time. What we ought to do is query once and provide access to the results of our collection. Let's do it!

CREATING CUSTOM COLLECTIONS VIA THE CONFIGURATION FILE

Let's go back to our DocPad Configuration File (`docpad.coffee`) and open it up. This time we want to add the following:

```
docpadConfig = {    collections:        pages: ->            @getCol
lection("html").findAllLive({isPage:true})}
```

In our default layout, `default.html.eco`, we'll update the `getCollection` line to become:

```
<% for page in @getCollection("pages").toJSON(): %>
```

Much better, and way more efficient.

Did you spot the difference with the call we used? When performing our query, we used the `findAllLive` instead of the `findAll` method. That's because `findAllLive` uses QueryEngine's Live Collections, which allows us to define our criteria once, and then continue to keep our collection up-to-date.

It works by creating a live child collection of the parent collection. (In this case, the `html` collection is the parent collection and our `pages` collection is the child collection.) The child collection then subscribes to the parent collection's `add`, `remove`, and `change` events, and tests the model that the event was for against our child collection's criteria. If it passes the collection, it adds it; if not, then it removes it. This performs much better than querying everything every single time.

So then, what about sorting? That's easy enough! We can sort by changing `@getCollection('html').findAllLive({isPage:true})` to add a second argument, which is the sorting argument; `@getCollection('html').findAllLive({isPage:true},[{filename:1}])` that, in this case, will sort by the filename in ascending order. To sort in descending order, we would change the `1` to become `-1`. Now we can sort by any attribute available on our models, which means that we could even add an `order` attribute to our document meta data and then sort by that if we wanted to.

There is also a third parameter for paging. To learn about this, as well as what type of queries are available to you, check out QueryEngine Guide.

SETTING DEFAULT META DATA ATTRIBUTES FOR OUR PAGES

Considering we'd probably like all our pages to use the default layout, we may be lazy enough to want to set this by default for all our pages, so we don't always have to add `layout: default` to the meta data of each page.

Just like everything, it's pretty darn easy, if you know how. And here's how:

```
docpadConfig = {    collections:        pages: ->            @getCol
```

So, what does this do? It's exactly the same as before, but we use the `add` event automaticaly fired by Backbone whenever a model (a page, file, document, whatever) is added to our collection. Then, inside our event, we say we want to set our default meta data attributes for the model; in this case, setting the layout to `"default"`.

This is invaluable when doing more complex things in DocPad. For instance, we use it for this documentation, to allow us to base the navigation structure of our documentation files on their physical location in our file system. Thus, if we have a file `docs/docpad/01-start/04-begin.html.md`, we can detect that the project is `docpad`, and assign `project: "docpad"` to the meta data accordingly. As the section is "start" and it is order first, we set `category: "start"` and `categoryOrder: 1`. We also see that our file is `begin` and ordered 4th. This is just one nifty example. There's plenty more you'll discover on your own epic journey! :)

ADDING THE BLOG POSTS

As you now have all the tools and knowledge required to be able to create the blog post section, we've left that part as an exercise for you! We've done this to help you retain and make best use of all the awesomeness you've just learned.

- If you need a few pointers to help you get started, here you go :)
    - Create a new layout called `post` that will use the default layout. Use it to perform custom styling for your blog post (e.g., `<div class="post"><%- @content %></div>`).
    - When creating your blog posts, we recommend giving them a `date` meta data attribute in the format of `date: 2012-12-25`, so you can sort your blog posts in descending date order.
    - Create a new directory called `posts` that contains all of your blog posts, and use the query `relativeOutDirPath: 'posts'` for your custom collection in order to retrieve all documents in the `posts` output directory (`/my-new-website/out/posts`). You can refer to the Meta Data Page for more information about the

    - Create a new page called `posts.html.eco` that lists all your blog posts. This will be, more or less, the same as our navigation menu. If you would like to display descriptons of the blog posts, just add that as a meta data attribute for the blog posts, and then output that meta data attribute. If you want to show the rendered content of the data, you can use `post.contentRenderedWithoutLayouts`. You can refer to the Meta Data Page for more information about the attributes already available to you.

- If you're stuck and need some help, just hop on over to the DocPad IRC Support Channel (#docpad on freenode) and someone will be with you soon enough. :) You can also discover all of our available Support Channels via our Support Page.

Congratulations! You now possess all the foundations required to be able to write amazing and powerful web applications like those already in ourShowcase. To recap, you now know how to:

- write documents in any language, markup, pre-processor, templating engine, whatever you wish, by installing the necessary plugin for it and changing the extensions of the document
- perform powerful abstractions using layouts, meta data, template data and configuration files
- create incredibly efficient custom collections, filtered and sorted by your own criteria
- do your own custom listings of content

But it doesn't stop there; these were just the foundations! If you can imagine it, then you will be able to accomplish it with DocPad. Really, there are no limits - that's why DocPad sets you free!

So, farewell and enjoy your epic journey. The power is yours!

DEPLOYMENT

Deployment is the next page of this guide.