Local Healthcare Application

Challenge: Create a application that helps individuals and communities to access (local) healthcare.

Solution: Medipath is a responsive and accessible app that offers medical solutions, healthcare and products to individuals.

Topics we'll be covering and what we've learned:

- Creating personas and user stories
- Sketching wireframes
- Affinity diagrams, themes and insights
- Prototyping
- Competitive audits
- Research studies

UX Design Thinking Process: 01. Empathize

The first stage (or mode) of the Design Thinking process involves developing a sense of empathy towards the people you are designing for, to gain insights into what they need, what they want, how they behave, feel, and think, and why they demonstrate such behaviors, feelings, and thoughts when interacting with products in a real-world setting

Personas

We started our project by empathizing with users and creating random personas that meets solutions for all users.

What are personas?

"Personas are fictional characters, which you create based upon your research in order to represent the different user types that might use your service, product, site, or brand in a similar way. Creating personas helps the designer to **understand users' needs, experiences, behaviors and goals"

Shima Qinyang

Beauty model

"Beauty is the illumination of our heart, body and soul"

Demography

Age:	27
Education:	Bachelor (Bsa)
Status:	Married
Location:	Shengzen, China
Carreer:	Beauty model

Home space — Office space

Solo — Collaborative

Mobile — Desktop

Psychographics

Motivation

Self-discipline

Sociable

Optimistic

User end goals

- *Expand family and have more children*
- *Beak free from trauma and local therapy visits.*
- *A dedicated online psychologist.*

Scenario

Shima is a successfull young model that was born and raised in Shenzen (China).
A adaptive and dynamic woman that is always on the go. She is married to Haoyu and they have 2 children.

Sadly, Shima experienced a dramatic trauma at a younger age, and often requires therapy for her mental health.

With her busy lifestyle, she hasn't much time to visit a psychologist. So she needs to find a way to do online classes.

Summary

Shima is a model that experienced a drastic trauma when she was younger. She has a quite busy lifestyle and is always on the move. Unfortunately she hasn't recovered yet from her past and requires therapy twice a week. She doesn't have much time and wants to discover a better alternative to therapy sessions.

Leo Muller

Teacher

"Education is a form of personal development"

Demography

Age:	38
Education:	High School
Status:	Single
Location:	Berlin, Germany
Carreer:	Unemployed

Home space — Office space

Solo — Collaborative

Mobile — Desktop

Psychographics

Motivation

Self-discipline

Sociable

Optimistic

User end goals

- *More information about healthcare documents*
- *Learn the dutch language*

Scenario

Leo had some personal issues in the past and recently moved to The Netherlands. He does not speak Dutch and is currently unemployed.

He is looking for a job but first needs to get his paperwork sorted out. He doesn't have a health insurance yet and doesn't precisely know where to find one. Because of the language, Leo cannot communicate with locals for assistance or whereabouts.

Eventually, Leo wants to learn the language and start a new life.

User stories

To better understand users we must create user stories based on the data of **personas.** This is so we can better recognize the user and their needs.

What are User stories?

A user story is a small, self-contained unit of development work designed to accomplish a specific goal within a product. A user story is usually written from the user's perspective and follows the format: “As [a user persona], I want [to perform this action] so that [I can accomplish this goal].”

SETTING UP MKDOCS APPLICATION OR WORKING ENVIRONMENT ON LOCAL MACHINE

MkDocs is a Python documentation tool that uses **Markdown** as its markup language to generate intelligent and beautiful documentation in **HTML**.

Building and testing the documentation source using Mkdocs on the local machine

To write and build the documentation website, you need to set up a Python virtual environment and install MkDocs.

Set up the Python virtual environment using this command:

```
python --m venv <folder_name(e.g: .virenv)>
```

Install MkDocs using this command:

```
pip install mkdocs
```

With MkDocs installed, you can run the command,

```
Create a new MkDocs project

mkdocs new <project_directory>
```

to create a new MkDocs project that contains a source directory (**docs**) and a default **mkdocs.yml** file with the most useful configuration values.

The **mkdocs.yml** contains the MkDocs configurations, where you can configure all aspects of how MkDocs reads your sources and builds your documentation.

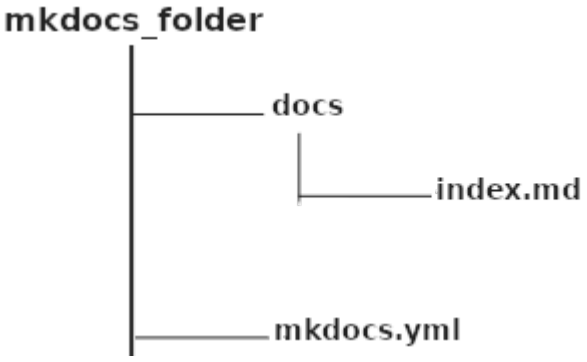

Fig. 1: Source directory for a MkDocs documentation

MkDocs reads its contents from files with the extension `.MD` which you have referenced in the **mkdocs.yml** file. These `.MD` files contain the structure of the documentation and the text to be displayed on the documentation website.

After you have set up the source directory for the documentation, you can use the command below to run the built-in development server provided by MkDocs.

Run the built-in development server

```
mkdocs serve
```

You can use the development server to test your documentation while building it. It is important to note that this command does not provide the documentation build files.

To generate the documentation build files, you can run this command:

Build the MkDocs documentation

```
mkdocs build
```

The command above builds the MkDocs documentation into a folder called **site**.

After writing and testing the documentation source on the local machine, you can transfer the source files (without the **site** folder) to your GitHub remote repository.

SETTING UP AUTOMATION USING GitHub AND GitHub ACTIONS

On the GitHub remote repository, we must set up GitHub Action workflows to handle automatic testing and building of the documentation whenever we trigger a push or pull request event.

What is GitHub Action?

GitHub Action is a (CI/CD) platform that allows you to automate your build, test, and deployment pipeline using workflows. When using GitHub Action, GitHub will provide you with Linux, Windows, and macOS virtual machines to run your workflows.

You can configure GitHub Action workflows to trigger when an event, such as a push, occurs in your repository.

Configuring a GitHub Actions Workflow

What is a workflow?

A workflow is a configurable and automated process that runs one or more jobs. Each job runs inside its virtual machine runner, or a container, and has one or more steps that either run a script you defined or run an action from the GitHub Marketplace.

Workflows are defined by a YAML file checked in your repository and are triggered either manually or by an event in the repository. Below is an example of a GitHub Action workflow.

```yaml
name: GitHub Actions Demo
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "🎉 The job was automatically triggered by a ${{ github.event_name }} event."
      - run: echo "🐧 This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo "🔎 The name of your branch is ${{ github.ref }} and your repository is ${{ github.reposi
      - name: Check out repository code
        uses: actions/checkout@v3
      - run: echo "💡 The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo "🖥️ The workflow is now ready to test your code on the runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "🍏 This job's status is ${{ job.status }}."

```

Fig. 2: A YAML file for a GitHub Action workflow

Using GitHub workflow to automate build and test process for our documentation

For us to build and test the documentation, we will configure two (2) GitHub Actions workflows and store them under the **.github/workflows** directory. The two (2) GitHub Actions workflows are **mkdocs_test.yml** and **main.yml**.

The **mkdocs_test.yml** workflow runs a CI test to check if the links in the documentation works. We trigger this workflow on each pull request events sent to the **main** branch of the official repository.

The **main.yml** workflow, on the other hand, check if the links in the documentation works, compiles the MkDocs sources in the official repository's **main** branch, and updates the **docs-build** branch with the build files. We trigger this workflow on each push events on the **main** branch of the official repository.

Testing and Publishing Documentation Changes

On the official repository, if a contributor sends a pull request to the repository's **main** branch, we test the changes by running the "*Pull Request MkDocs Check*" (i.e., mkdocs_test.yml) workflow.

If the test is successful, then the documentation project maintainer will merge the pull request changes after reviewing the pull request. This is to ensure that we review the changes in the pull request before merging into the official repository.

When the documentation project maintainer merges the pull request changes to the repository's **main** branch, GitHub triggers the "*Compile MkDocs source and update docs-build branch*" (i.e., main.yml) workflow automatically to build the documentation.

If the build is successful, it sends the documentation build files to the **docs-build** branch. The **docs-build** branch is where the hosting platform copies the documentation build files to update the documentation website.

MAINTAINING DOCUMENTATION

Creating the structure of a document

Markdown lets you add structural elements to your document, such as **headings** (`h1` , `h2` , `h3` etc.). The hashes move lower-level headings further to the right, so they appear indented. There are a few ways to add headings in Markdown. The recommended one is to prefix a heading with hashes **#**, one for each level of heading:

```
# Heading 1
## Heading 2
### Heading 3

And this is a paragraph.
```

Sections of a document can be separated using **horizontal rules** (`<hr />`), or lines. You create these in Markdown using three (or more) hyphens `-`, asterisks `*`, underscores `_` or equals `=` signs. Place them alone on a line, with blank lines on either side:

```
Brief introduction.
===
# Chapter 1
Lots of text.
---
# Chapter 2
Some more text
---
```

Lists are another important structural element. Unordered lists (`<ul>`) are created by beginning the line with an asterisk `*`, plus `+` symbol, or hyphen `-`, followed by a space or tab, then the text.

Ordered lists (`<ol>`) are numbers followed by periods. The numbers don't necessarily have to be in order. Below is an example of an unordered and ordered lists

```
### Unordered List
* this is an
* unordered list

+ this is another
+ unordered list

### Ordered List
1. this is an
2. ordered
3. list

1. and so
1. is this too
```

Note

If you want to start a line with a number and a period without starting a list, you need to escape the period with a backslash \:

```
2020\. A year we'll never forget.
```

Finally, paragraphs of normal text are separated by one or more blank lines:

```
This will be formatted as an HTML paragraph.
```

Starting a new document

MkDocs uses regular Markdown (`.md`) files as the source for its documentation. We place these Markdown files in the documentation directory called **docs** which exist at the top level of your project, alongside the **mkdocs.yml** configuration file.

All Markdown files included in your documentation directory will be rendered in the built site, regardless of any settings.

The simplest project you can create will look something like this:

```
mkdocs.yml
docs/
    index.md
```

You can also create multipage documentation, by creating several Markdown files:

```
mkdocs.yml
docs/
    index.md
    about.md
    changelog.md
```

The file layout you use determines the URLs that are used for the generated pages. Given the above layout, pages would be generated for the following URLs:

```
/
/about/
/changelog/
```

You can also include your Markdown files in nested directories if that better suits your documentation layout.

```
docs/
    index.md
    user-guide/getting-started.md
    user-guide/configuration-options.md
    changelog.md
```

Source files inside nested directories will cause pages to be generated with nested URLs, like so:

```
/
/user-guide/getting-started/
/user-guide/configuration-options/
/changelog/
```

After creating the Markdown files in your documentation directory, you must configure pages and navigation in the **mkdocs.yml** file.

The `nav` configuration setting in your **mkdocs.yml** file defines which pages are included in the global site navigation menu, as well as the structure of that menu. If not provided, the navigation will be automatically created by discovering all the Markdown files in the documentation directory.

A minimal navigation configuration could look like this:

```
nav:
    - 'index.md'
    - 'about.md'
```

or

With user-defined titles

```
nav:
    - Home: 'index.md'
    - About: 'about.md'
```

After configuring pages and navigation, you can test the documentation by executing the command below to start the built-in development server:

Run built-in development server

```
mkdocs serve
```

> **Note**
>
> The development server will not start successfully if there is an error in the source files.

Updating an old document

Updating an old document is easy. You just have to find the old document (`.md`) file and make the necessary changes to it. You must ensure the configuration file (**mkdocs.yml**) is intact.

After making the changes, execute the command below to start the built-in development server:

Run built-in development server

```
mkdocs serve
```

> **Note**
>
> The development server will not start successfully if there is an error in the source files.

Serving images in a document

In MkDocs, images are served from the folder in the **docs** directory. You can then link an image in a source file by using the relative path to that image.

Note

It is not compulsory to store the images under the **img** folder. You can decide to store your images in any folder, but the folder should be in the **docs** directory.

The code below shows how to add an image using the Markdown syntax:

```
<![Alt text](relative_path_to_image)>
![Example of an image](img/example.png)
```

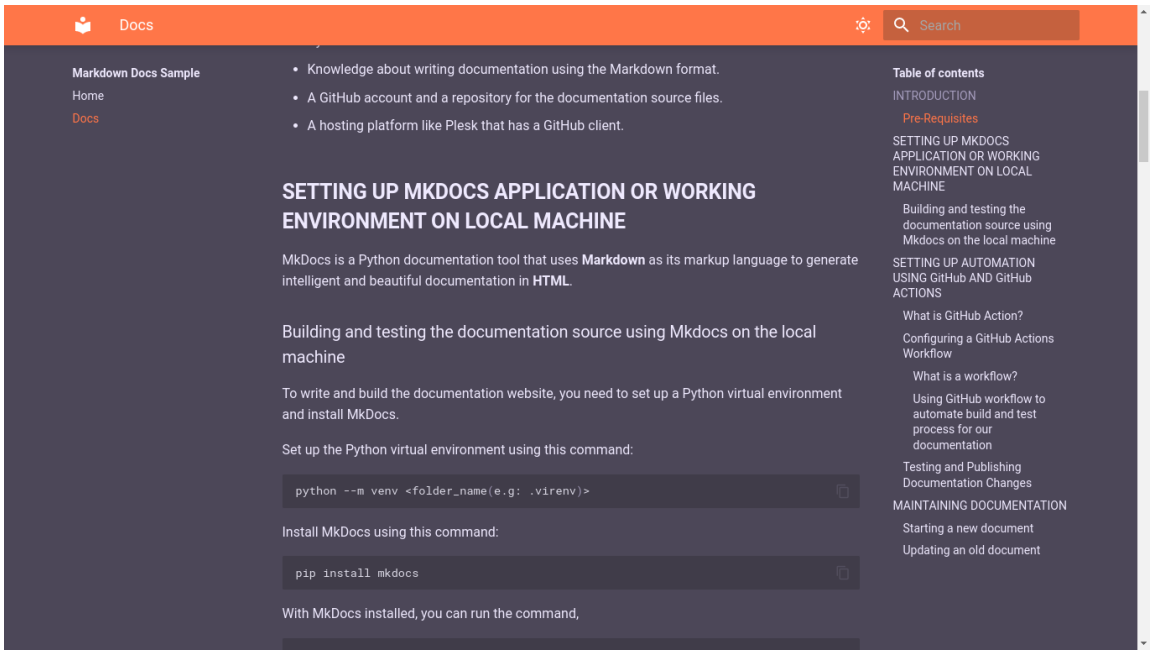

Fig. 3: Output of the code above

Tip

To properly maintain images for large documentation sources, it will be appropriate to divide your images into parts and store them in separate sub-folders under the **img** folder. For example, all images for the homepage should be stored in the folder called *homepage* and images for the about section should be stored in the folder called *about*.

How to generate a PDF for a specific page in the documentation

To generate a PDF for a specific page, you need to add the MkDocs PDF Export Plugin to your MkDocs project.

The MkDocs PDF Export Plugin is a plugin to export content pages as PDF files. Before installing the plugin, you need to have some packages, which the plugin's documentation explain.

Install the package with pip:

```
pip install mkdocs-pdf-export-plugin
```

Enable the plugin in your `mkdocs.yml`:

```
plugins:
    - search
    - pdf-export
```

When you build the documentation, you will see a download button, as described in Fig. 4 below, for every page in your MkDocs project.

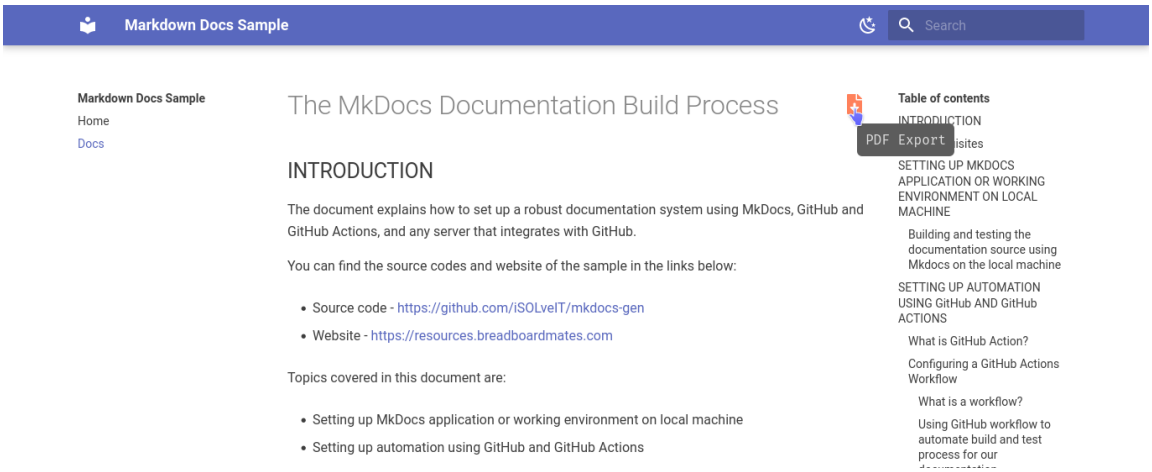

Fig. 4: PDF download button for index page