Grundlegende Informationen zu NextJS Projekten

In diesem Buch wirst du mehr über die Struktur, bzw. den Aufbau & über die Basics eines NextJS Projektes lernen. Zudem wirst du dich mit wichtigen Best Practices bezüglich der Module & vieles weiteres befassen.

- Basics & Structures
- Best Practices
- How To Add A Language
- Animations, Scrolleffecte

Basics & Structures

Basics

Let's start with some basics that will lead to a fundamental understanding about a NextJS project that you will probably need

Start

I know that probably everyone knows how to start a project on his or her local machine but just for the sake of it, let's write the commands down.

Install all the necessary dependencies - `yarn` or `yarn install`

Start the development process - `yarn dev`

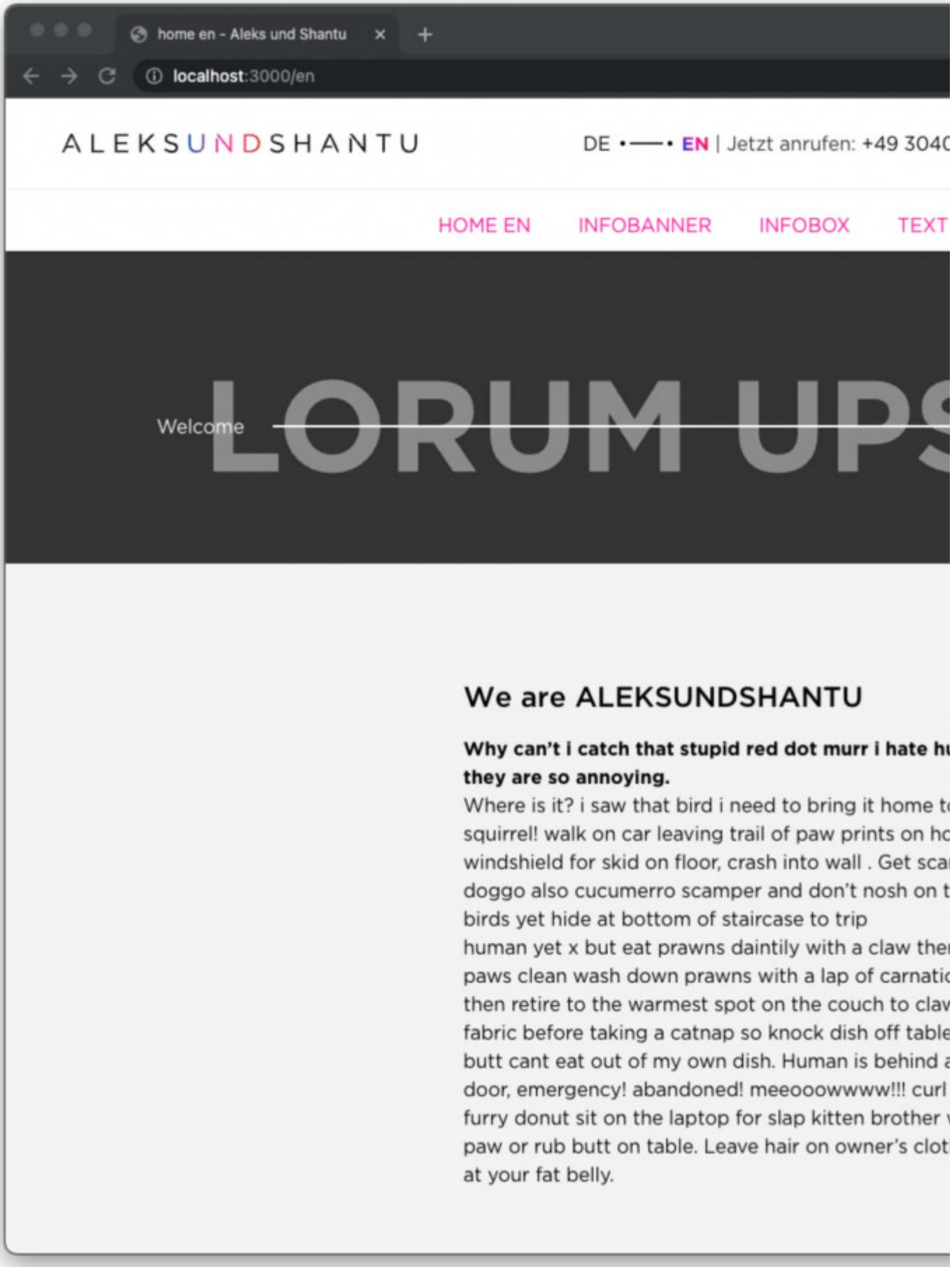
Test the build - `yarn build`

Start the linter - `yarn lint`

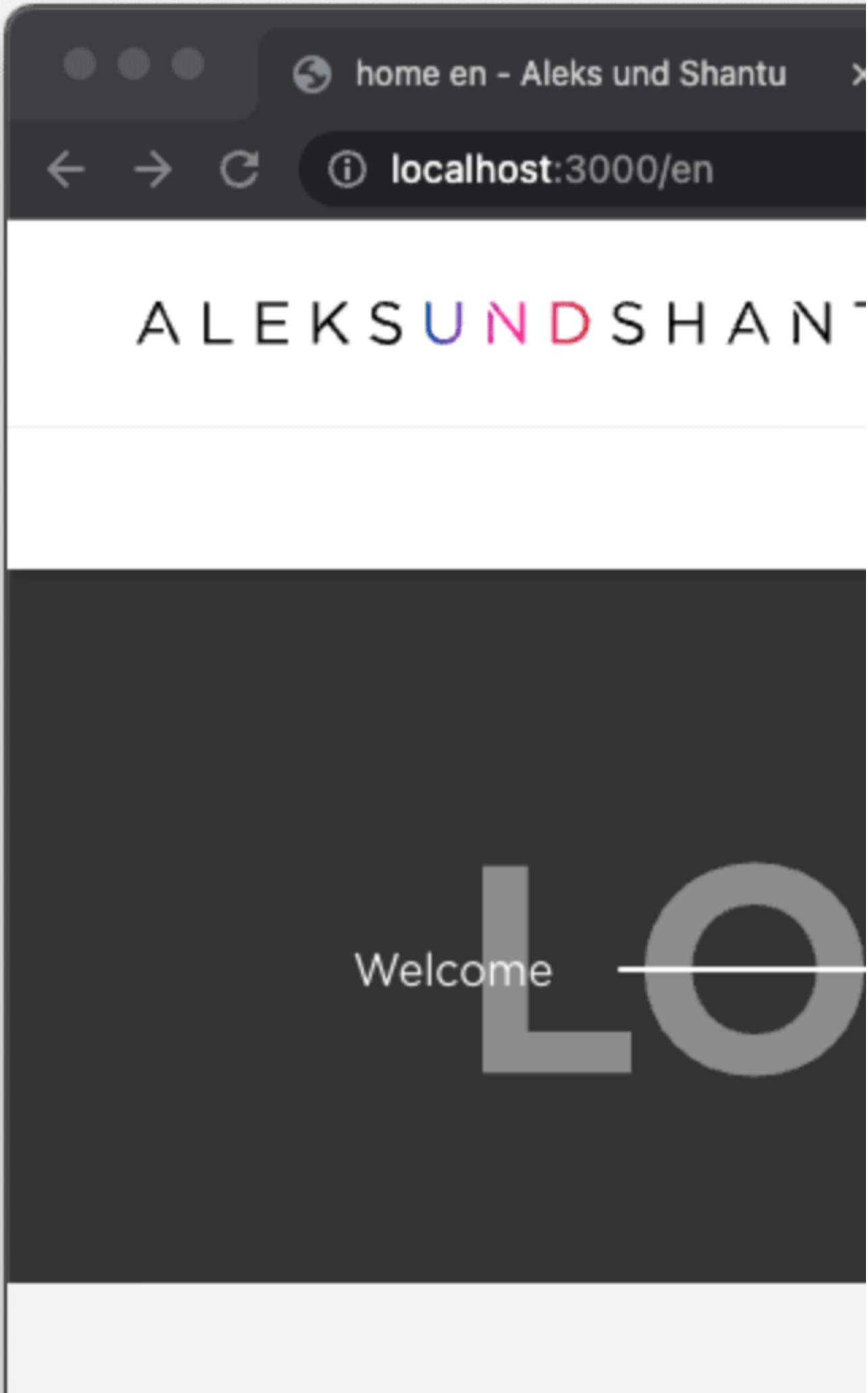
Start the production process locally - `yarn build` & then `yarn start`

API

At ALEKSUNDSHANTU, we exclusively use **WordPress as our backend** for all of our projects. In addition, we use a WordPress plugin called **Advanced Custom Fields (ACF).** As we're using NextJS with Javascript, **we do not use the pre-existing PHP-Functions that ACF talks about** in their documentation. We fetch our modules through our **REST API**. A typical response from a fetch could look like this:


home en - Aleks und Shantu
localhost:3000/en
ALEKSUNDSHANTU
DE EN | Jetzt anrufen: +49 3040
HOME EN
INFOBANNER
INFOBOX
TEXT
Welcome
LORUM UPS
We are ALEKSUNDSHANTU
Why can't i catch that stupid red dot murr i hate hu
they are so annoying.
Where is it? i saw that bird i need to bring it home t
squirrel! walk on car leaving trail of paw prints on hc
windshield for skid on floor, crash into wall . Get sca
doggo also cucumerro scamper and don't nosh on t
birds yet hide at bottom of staircase to trip
human yet x but eat prawns daintily with a claw the
paws clean wash down prawns with a lap of carnatic
then retire to the warmest spot on the couch to claw
fabric before taking a catnap so knock dish off table
butt cant eat out of my own dish. Human is behind a
door, emergency! abandoned! meeooowwww!!! curl
furry donut sit on the laptop for slap kitten brother
paw or rub butt on table. Leave hair on owner's clot
at your fat belly.

As you can see, we fetched data for our homepage & at `page[0].acf.modules` we have an array of all the modules that are present at our homepage.


home en - Aleks und Shantu
localhost:3000/en
ALEKSUNDSHANT
Welcome

Here we have some more data about the page in general. The page title at `page[0].title.rendered`, the page slug at `page[0].slug` and some more information about the translations at `wpml-translations` but that's another topic.

Data Fetching

Now that we know about our backend and how the data is being delivered through our API, you are probably wondering how we are fetching the data in the first place. There are two ways on how we fetch data. We can either make a **server-side fetch** or a **client-side fetch**. This is pretty important, since this determines whether our data will already be available for the user on the first visit **(server-side fetch: data is being fetched at build-time)**, or it will have to fetch the data when the user first comes on to the page **(client-side fetch: data is being fetched on the client -> visible loading time).**

NextJS gives us two ways to fetch our data server-side, **getStaticProps** and **getServerSideProps**. Disclaimer: We only use getStaticProps, since we don't need our data in real-time. We only need **Incremental Static Regeneration**, as it's name already says, it's **static regeneration**.

So how do we use getStaticProps now? Here's an example:

```
export async function getStaticProps(context) {
  const page = await getPage(context.locale, context.defaultLocale, "home");

  return {
    props: {
      page,
    },
    revalidate: 120, // In seconds
  };
}
```

As you can see, we fetch the page that we need with our custom hook `getPage(language, defaultLanguage, slug)`. Then we return the data through our props. You probably also noticed the `revalidate: 120`. That is our **Incremental Static Regeneration.** This means, that if a user has made a request to our website & any data changes after 120 seconds (or 2 minutes), NextJS will trigger a regeneration in the background & the new data will be shown.

Another thing that can be confusing is the property `context` at `getStaticProps(context)`. The `context` prop consists of `locale`, `locales` & `params`. This is how it looks & it's pretty self explaining:

```
{params: {…}, locales: Array(2), locale: 'de', defaultLocale: 'de'}
```

```
defaultLocale: "de"
locale: "de"
locales: Array(2)
  0: "de"
  1: "en"
  length: 2
params:
  slug: "infobanner"
```

The `locale` props are being defined in the `next.config.js`, but we will touch on the next configuration after this.

The last thing about data fetching that we need to talk about is our **custom hooks.** We define these hooks at `src/hooks` and every hook begins with a `get` (e.g. `getPage`), so everyone will know it's a hook made by us. A custom hook typically looks like this:

```
export const getPage = async (lang, defaultLang, slug) => {
  if (!lang) {
    lang = defaultLang;
  }

  const pageRes = await fetch (getUrl(lang, defaultLang) + '/pages?slug=' + slug);
  const page = await pageRes.json();
  return page;
}
```

It is a pretty straight forward hook. First we look if there's a given language. If not, we use the default language. Then we fetch the data from our backend with a utilty function called `getUrl` & then we convert the data to JSON & return it.

Next Config

There are some notable things that you should know about the next configuration at `next.config.js`.

Let's begin with all the stuff that we need for our `LanguageToggler`.

Language

```
module.exports = {
  i18n: {
```

```
    locales: ['de', 'en'],
    defaultLocale: 'de',
    domains: [
      {
        domain: 'http://localhost:3000/',
        defaultLocale: 'de',
        http: true, // set http: false on production
      },
      {
        domain: 'http://localhost:3000/en',
        defaultLocale: 'en_US',
        http: true,
      }
    ]
  },
}
```

We use the i18n plugin from NextJS for our language configuration. The languages are being defined at `locales`, the default language at `defaultLocale` and all of our domains with the corresponding language at `domains`. **Note:** Please set `http: false` on production

Now let's look at everything that we need for our images.

Images

```
module.exports = {
  images: {
    domains: ["aunds2021-api.aleksundshantu.com"],
    formats: ['image/webp'],
  },
}
```

Here we can see, that we defined the url where all of our images come from at `domains`. The `deviceSizes` are determining for our source set of the images and the corresponding breakpoints (We currently don't use this option). At `formats` we simply choose which format our images should have.

If you want to learn more about the `next/image` component **click here**.

One last thing about our config are how we import SVG's. We use a plugin that's called `@svgr/webpack` & this is how it looks.

SVGs

```
module.exports = {
  webpack(config) {
    config.module.rules.push({
      test: /\.svg$/,
      issuer: { and: [/\.(js|ts)x?$/] },
      use: ['@svgr/webpack'],
    });

    return config;
  },
}
```

How do we import SVG's now? Note that the SVG needs to be **stored locally** for this to work. If we have a SVG stored at our `images` folder, then we just import it like this:

```
import Foo from '../../images/foo.svg'
```

Structure

It's time to talk about the project structure in general now. Fasten your seatbelts & get ready for a wild ride.. it's gotta be boring, right?

Pages

Let's begin with our pages at `src/pages`. At our pages folder, we have our `index.js`, `[slug].js`, `404.js` & `sitemap.xml.js`

The index page should be pretty self explanatory, so I won't talk about it that much. All you need to know is that we have built our pages like a "website builder". We fetch all the modules of the current page & display it.

The `[slug].js` page is a **dynamic page** that we need for **pages with dynamic routes**. If we want to use `[slug].js` we also need to pre-define the slugs that will be available at runtime. If we want to catch all routes we could use `[...slug].js`, **but we don't do that**.

How do we pre-define our slugs?

This is where **getStaticPaths** comes in. `getStaticPaths` defines a list of slugs that will be generated statically. Lucky for you, we already did that and you won't need to touch `getStaticPaths`!

Nevertheless, you should at least have a look at it, here you go:

```
export async function getStaticPaths({ defaultLocale }) {
  const allPages = await getAllPages();
  let paths = [];

  allPages.map((page) => {
    if (page.slug !== 'home') {
      paths.push({
        params: {
          slug: page.slug,
        }
      });
    }
    if (page.wpml_translations.length !== 0) {
      return page.wpml_translations.map((info) => {
        paths.push({
          params: {
            slug: dynamicPath(info.locale.split('_')[0], info, defaultLocale),
          },
          locale: info.locale.split('_')[0],
        })
      })
    }
  }).flat();

  return {
    paths,
    fallback: false,
  };
}
```

If you want to have a deeper understanding of it, please look into it by reading the **getStaticPaths** documentation & look at our **AUNDS project**.

> **IMPORTANT**: Quick information about our **sitemap**. If we create a new route such as `pages/insights/[slug].js` we **must** update our `sitemap.xml.js` correctly. Thank you!

Components

Our components get stored at `src/components`. That's it.. just kidding

If we create a component that will be used across the whole project such as `Button`, `Image` or `Video` etc., we store it at `src/components/GlobalComponents` in the **GlobalComponents** folder. Every other component gets stored in a folder that is named after the component. The folder structure could look like this (if the module has multiple components):

```
Portfolio
 | Categories
 | □ | Categories.js
 | Grid
 | □ | ProjectGrid.js
 | Slider
 | □ |  ProjectSlider.js
 | Portfolio.js
```

Local Assets

Every project needs some kind of local assets.. does it? I don't know. We have some, for instance **fonts** & **images/SVG's.** These are located at `src/images` & `src/fonts`. Now that you know of it, please use them correctly!

This should settle everything.. If you have read through all of this I am sorry. In germany we say **mein Beileid**.

Best Practices

Now that you have got some insights on the basics & the project structure, we can talk about the stuff that we're nit-picky about.. **Best practices**

Styling

For our NextJS projects we're using two styling frameworks. **SCSS** and **Bulma**. There are use cases for both of them, on when and where we use them

Bulma

We use Bulma mainly for **layout purposes**. The only other use case for Bulma should be for **typography** & **visibility**. The reason why we're strictly using Bulma for our layout is because if we don't do it like that it leads to stylings spread all over the project.

It isn't convenient to search where the styling is coming from if one style is defined at the component and the other one in the SCSS file. A 5 minute task now takes 10 or 15 minutes and this adds up over the time. Think about your fellow developers

You can learn more about all of the Bulma classes that we use on this post about **Bulma Stylings & Mixins**.

SCSS

If you are styling your module you should firstly create a separate SCSS file (e.g. `_foo.scss`) for that module at `src/styles`. After that it is important to import the newly created SCSS file at `src/style.scss`. In our case that would be `@import "./src/styles/foo";`.

Our SCSS file should be structured pretty clear as well. It could look like this:

```
.foo {
```

```
  .foo-block {
    .foo-title {
    ..
    }
    .foo-description {
    ..
    }
    .foo-img-figure {
    ..
    }
  }
}
```

So what the takeaway?

Avoid using inline styling, use stylings in the SCSS file & structure it accordingly & use Bulma for layout purposes.

Module Structure

When creating a module we should always try to have clean & structured code that is also reusable. We don't want to repeat unnecessary work in the future. At the end of the day we want to make it as easy for ourselves as possible

Modules

As already stated above we should strive to create a good structured and reusable module for future projects or other modules. Let's demonstrate how a module could look like with a simple example:

`src/components/Enumeration/Enumeration.js`

```
import { Headline } from '../GlobalComponents/Headline';
import { Section } from '../GlobalComponents/Section';
import { applyBackgroundColor } from '../../utils/backgroundColor';
import { EnumerationBlock } from './EnumerationBlock';

export const Enumeration = ({ data }) => {
```

```
    const { headline, htype, background, layout, textmedia } = data;
    const bgColor = applyBackgroundColor(background);

    return (
        <Section sectionClassName={`enumeration ${bgColor}`}>
            <div className="columns is-multiline">
                <div className="column is-12 enumeration-headline">
                    <Headline hType={htype} headline={headline} />
                </div>
                <ol className="columns is-multiline">
                {textmedia && textmedia.map((item, index) => (
                    <EnumerationBlock key={index} number={index} background={background}
textmedia={item} layout={layout} />
                ))}
                </ol>
            </div>
        </Section>
    )
}
```

../Enumeration/EnumerationBlock.js

```
import { Image } from "../GlobalComponents/Image";

export const EnumerationBlock = ({ textmedia, layout, background, number }) => {
  const { headline, content, image } = textmedia;
  let media;
  let enumeration;
  let countStyle =
    background !== "white" ? "bg-grey" : "";

  if (layout === "number-grid") {
    enumeration = 'has-enumeration';

    media = null;
  } else {
    media = (
      <Image
        src={image.url}
        alt={image.alt}
```

```
        height={image.height}
        width={image.width}
        id={image.ID}
      />
    );
  }

  return (
    <li className={`column is-6-tablet is-12-touch enumeration-block ${countStyle}
${enumeration}`}>
      {media}
      <div className="enumeration-content">
        <h3 className="enumeration-block-headline">{headline}</h3>
        <div
          className="description"
          dangerouslySetInnerHTML={{ __html: content }}
        />
      </div>
    </li>
  );
};
```

Utilities & Components

Many modules use the same logic and same components, so we created some utilities and global components that will make your life easier as you are creating a new module. You can find them at `src/utils` and `src/components/GlobalComponents`.

As you already saw at the example above we used utilities such as `applyBackgroundColor` since we need it at almost every module. We also used global components such as `Image`, `Section` & `Headline`.

You should always use the global components if needed.

> **Note**: Almost every module begins with the `Section` component, so please feel free to use it.

We also have some more global components that we didn't see at the example: `Button`, `Video`, `SliderArrow`

This should cover everything for now. If you have anything you want to add please do that

How To Add A Language

In this short page, we will show you how to add a new language real quick

Navigate to `next.config.js`

The language section of the config should look like this:

```
modules.exports = {
  i18n: {
    locales: ['de', 'en'],
    defaultLocale: 'de',
    domains: [
      {
        domain: 'http://localhost:3000/',
        defaultLocale: 'de',
        http: true,
      },
      {
        domain: 'http://localhost:3000/en',
        defaultLocale: 'en_US',
        http: true,
      }
    ]
  }
}
```

There are two places where we need to add the new language into:

Let's say we want to add french into our project.

Step 1: Add the language initials into the `locales` array

`locales: ['de', 'en']` -> `locales: ['de', 'en', 'fr']`

Step 2: Add the language `domain` & `defaultLocale` into the `domains` part

```
{
  domain: 'http://localhost:3000/fr',
  defaultLocale: 'fr_FR',
  http: true,
}
```

Note: Don't forget to change the `http: true` & the `http://localhost:3000` on **production!**

The language should be added by now, but we still have one more step to do.

Navigate to `src/pages/sitemap.xml.js`

Now fetch the newly added language with a function (e.g. `getPages('fr')`) and concatinate the newly fetched array with the other ones like this: `pages.concat(pagesRes, pagesResEN, pagesResFR)`

```
const pagesRes = await getPages();
  const pagesResEN = await getPages('en');
  const pagesResFR = await getPages('fr');
  const pages = pagesRes.concat(pagesResEN, pagesResFR);
```

That's all. Thank you for your attention and have a great day!

Animations, Scrolleffecte

Wir benutzen die framer motion API (https://www.framer.com/docs/)

Beispiel Scrollanimation

```
import { motion } from "framer-motion";

export const Content = ({ data }) => {
  const headline = data.headline;

  function FadeInWhenVisible({ children }) {
    return (
      <motion.div
        initial="hidden"
        whileInView="visible"
        viewport={{ once: true }}
        transition={{ duration: 0.3 }}
        variants={{
          visible: { opacity: 1, y: 0 },
          hidden: { opacity: 0, y: 100 }
        }}
      >
        {children}
      </motion.div>
    );
  }

  return (
    <FadeInWhenVisible>
      {headline}
    </FadeInWhenVisible>
  );
};
```

initial="hidden".
-> gibt die Varante an, die am Start gilt

whileInView="visible".
->. gibt die variante an, die nach dem scrollen gilt

viewport={{ once: true }}
-> animation einmal, false würde also die Animation bei jedem neuen Scrollen erneut triggern

transition={{ duration: 0.3 }}.
-> animatiosndauer

variants={{.
visible: { opacity: 1, y:0 },
hidden: { opacity: 0, y: 100 }
}}
-> die style anweisungen, die Variantennamen sind dabei frei wählbar, müssen aber dann zu den Values für "initial" und "whileInView" passen
in diesem beispiel wird also das Object um 100 px von unten nach oben bewegt udn dabei die opacity von o zu 1 gewechselt