Apprenticeship Portfolio

Founders & Coders

September 2021 - November 2022

[COMPANY NAME]

About the website

`[COMPANY NAME]` is the company's own website. We recently revisited the project and gave it new look and content updates. There is a showcase of our most recent and popular projects, as well as info about the team, our approach, etc.

Scope of my given tasks 👓

I took up a few different tasks in the creation of the website. I was involved in the front and back end, focusing on building, styling and testing individual components. The main technologies I used were `Next.js` and `Stitches` in the front end and `Sanity CMS` for the back end. For visual regression testing, I used `Storybook` and `Chromatic` in continuous integration.

Software Development Lifecycle Stages

Planning

Roles the team took on for the project

Our team in [COMPANY NAME] consists of **designers**, **producers** and **developers**. Each one has a distinct role in their goals and the tasks they have to carry out.

The designers work in most stages of a project: the conception of the idea, the prototypes and wireframes, the core user stories, designs, typography and layouts.

As this is the second iteration of this project, we already had many elements available, like the brand, representative colours and company values. This time we also had lots of projects to talk about, as well as an expanded team. We focused on putting all these things together in a storytelling manner and presenting our work in an engaging way.

Sprint structure & build order

This was an internal, medium size project that we were working on in parallel with client work. Some hours were allocated for it every week, and the goal was to launch the new version as soon as possible during the 2022 summer.

The `Agile` approach was better suited as a development methodology in the case of this project. In particular, the goal for the development team was to create as many tickets as possible in a granular way, so that anyone with availability could pick anything up. As soon as a working feature was built and reviewed, it was added to the codebase no matter how small or big.

We tried to avoid the more solid structure of a waterfall approach, considering that the available time was not secure in a consistent way. The agile approach allowed us some flexibility and since we had all our epics and main stories ready, we could work on them in a distributive fashion inside the team.

Analysis

During the discovery phase, we consider different options regarding the technologies we would use. As we are a digital studio, our projects are pretty media-heavy and the visual aspect is very important. `Next.js` is usually our preferred framework for the front end since it offers many out-of-the-box optimisations and the deployment process is quite straightforward with `Vercel`. There were a few other things that needed consideration though.

I completed a code planning exercise with the company's senior developer. Here we looked at the software requirements and how it affects different parts of our stack.

Our primary considerations were:

- **Organising architecture** in an effort to scale up the product. For example, showcasing the projects is the highlight of the website, and we should be able to add more in the future in a simple and intuitive way.
- Presenting our material in an equally **visually appealing** way on desktop and mobile. Which styling framework would offer flexibility in doing so?
- Identifying **common design patterns**, so we can divide the work into functional components that can be reused.
- Effectively **storing our content**. Although we initially planned for this to be a static website, we quickly realised that storing content in the codebase would be highly impractical. Whenever we would like to add a new project, update info about the team or anything else relevant, a developer would need to

modify the codebase.
To make things more flexible and accessible to everyone in the team, we added we decided to use `Sanity CMS` for content management.

User research findings

We conducted user research internally and tried to put the collective experience of the team into action. This was particularly useful in helping me understand the needs of the project better in terms of brand, delivery and functionality.

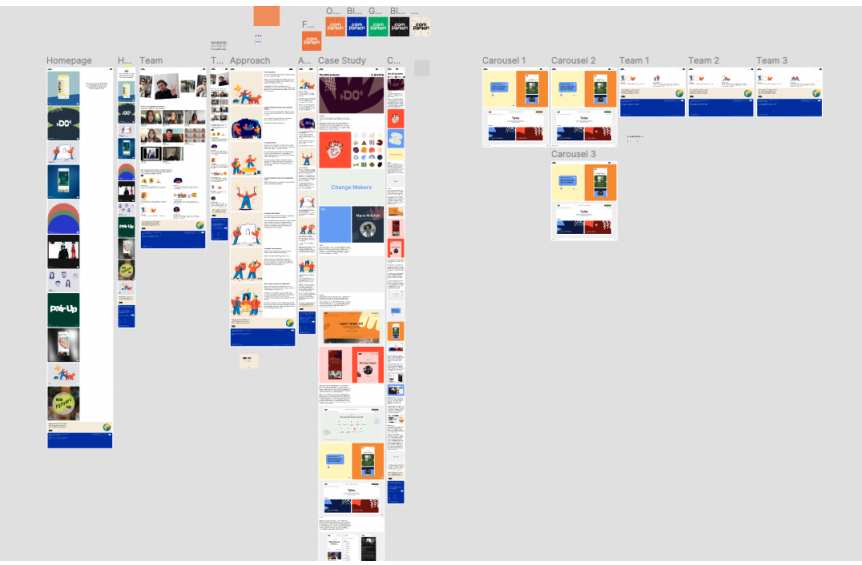
Some of the main points that were made:

- We need to **communicate our work** with potential clients/collaborators. **Showcasing** our products in an engaging and informative way that shows the true spirit of the team.
- Increase our presence on **social platforms**. The website should be performant SEO-wise.
- Demonstrate our **skill as a team** to take care of all different parts of a product and deliver high-quality results.
- Present ourselves as neutral in terms of **style**. We can mould our stylistic approach according to clients' requests and thus present a range of variety.

The general consensus was that we want an [CLIENT ONE]ed website, where the main characters are the projects. We do not need fancy visual effects which could distract from the material, but instead, clear structure and sections. We ended up with three main pages: one for the projects, one for the team and one for the company approach and values.

Design

Here are some first ideas using the artwork with the company's featured characters that showcase potential layouts, colours, typefaces etc.



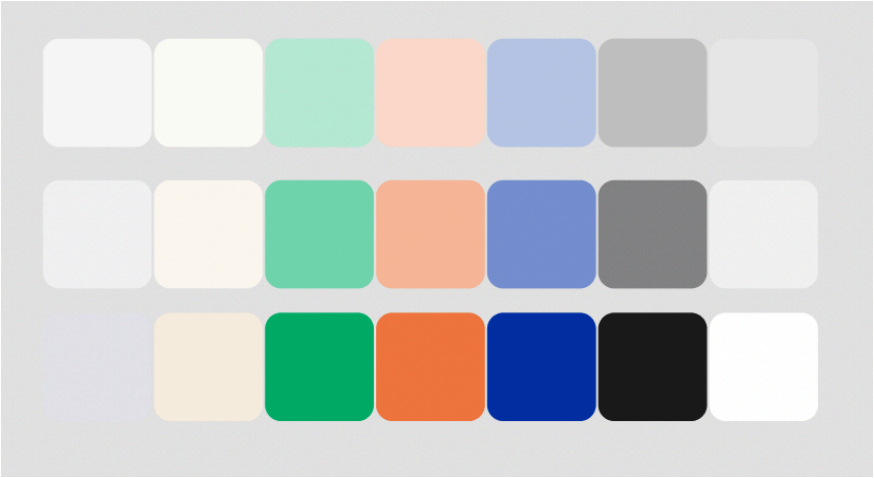
The main characteristics were:

- **accessibility** - the users should be able to easily navigate the website and access all sections with any input device.
- **[CLIENT ONE]ed interface** - the interface should be simple and clear, with no hidden flows. The company projects are in the spotlight as soon as somebody lands on the page.
- **aesthetically pleasing** - the website should communicate professionalism and confidence, but be approachable and warm. We do not create complex methods of consumption and when we use animation or any interaction, it should serve the purpose of improving the user experience.

Colours and [COMPANY NAME] mascots

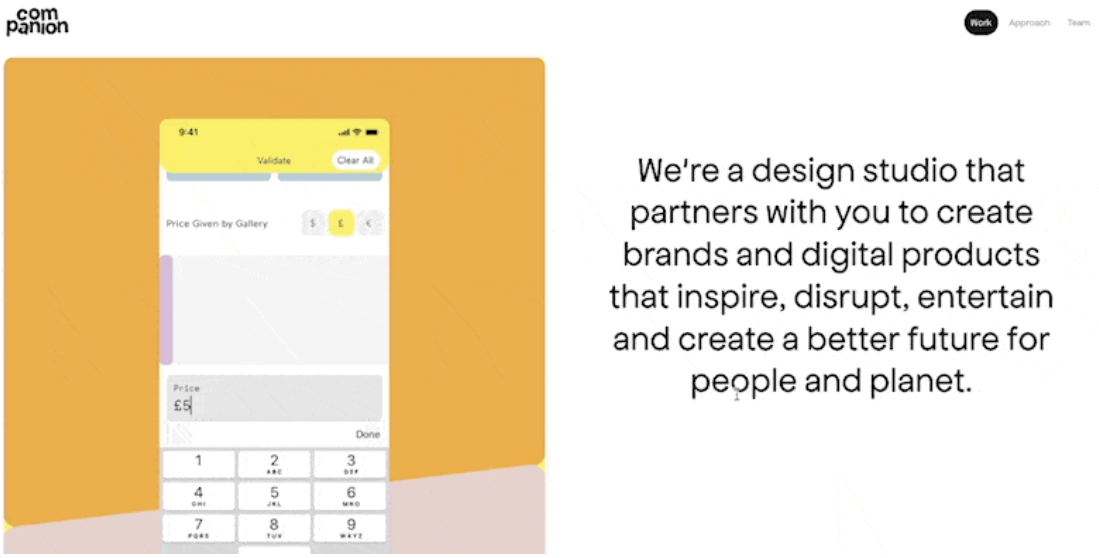
Based on all considerations described above, this became the chosen colour palette.

These colours have been part of the [COMPANY NAME] brand since the beginning and have been used in the illustrations that feature the studio characters: the dog, bird, turtle and fox. These figures represent the values we share collectively as a team about collaboration, trust and creativity.



The “Homepage”

The homepage is simple and effective. On desktop, all completed projects appear on the left side, while the company’s statement always appears on the right. On mobile, the statement stays on top and the projects follow.

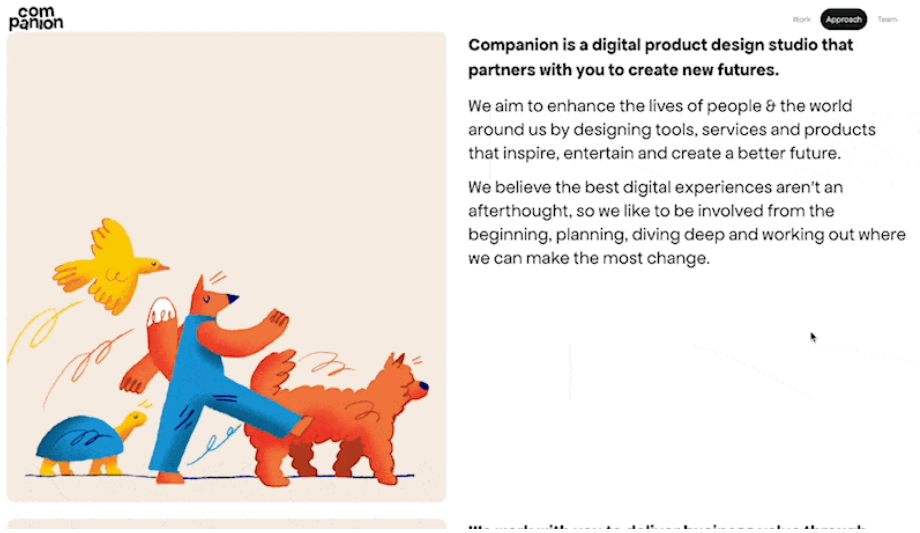


The Homepage

Navigation and other pages

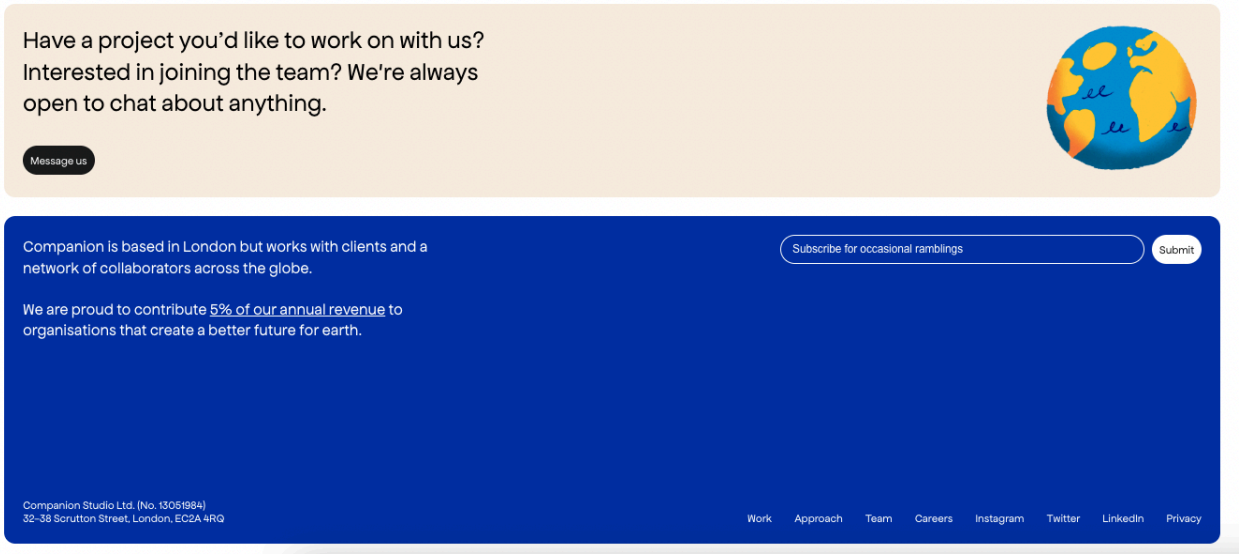
The user can navigate the other pages through the menu in the navigation bar.

The general structure of the pages is quite homogenous, with lots of visual assets and intersecting text that provides the context. Much of the content one can see in the “home” and “approach” pages comes from the same components.



The Approach page

Lastly, the footer summarises useful links and a newsletter that is prompting the user to keep in touch.



CTA & footer

Build

Technical decisions

The major technical choices we had to make based on the nature of the website were the following:

- Where should we store the content? Is the website going to be big enough to justify using CMS? In fact, when I started building the first components, all data was static, stored in a javascript file along with the rest of the codebase. The number of finished projects was not that large yet, and the variety was still limited. But the project number kept growing and showcasing them became our main focus. We wanted to add flexibility to how we manage an existing page and how we add more projects, delete old ones, etc. This is why we decided to use a headless cms, `Sanity CMS`.
- For the front end, we went with `Next.js`, which combines all the advantages of the `React` library, while at the same time offering server-side rendering. That means that it combines quite well with the headless cms since it can communicate with data coming from some sort of database, make requests to it and retrieve them server-side. At the same time, it can statically generate websites. All these qualities make it more performant compared to React.

Stack I am using

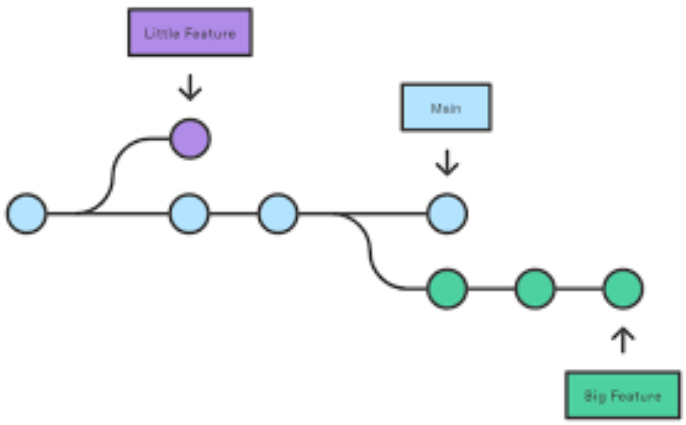
Here is a list of main technologies we are using in the project:

- `Next.js` for the front-end and server-side rendering
- `Sanity` for content management (CMS)
- `Storybook` with `Chromatic` for designing components in isolation and visual regression testing

Version control system and collaboration

As in all projects, we are using `Git` as a version control system. This allows us to manage and keep track of the source code of the project. The project repository is hosted on `GitHub`, a cloud-based hosting service for managing `Git` repositories.

Each developer works on a separate branch of the `main` one when working individually. A branch that contains completed work and has been reviewed by the team is getting merged into `main` and becomes part of the codebase.



Deploy

The website is being deployed in `Vercel`, a platform built by the `Next.js` creators. It offers seamless integration with Next applications, as the deployment happens automatically with every push to a GitHub branch, which can be easily merged into production.

Having automatic deployments for both development and production environments is extremely helpful, as we can easily check and share new features, without extra work overhead. Among many other features, Vercel offers `Environment Variables` storage, and integration with external services like `Doppler` (for environment variables management) and `GitHub`, as well as custom domain names.

Maintain

The project is currently considered completed as all main and extra features have been built and the website is launched. We have the flexibility to add new projects we want to showcase since the content is dynamically managed with Sanity CMS. That means that no development work is needed unless our testing suite tracks an error or visual regression or a new feature is designed in the future.

Examples & code evidence

Building the `Sanity` schema

At the first stages of building the application, we decided to implement the front end and back end separately, since some of the designs were not yet agreed upon at the time, although the structure of the content was known.

I started creating the `Sanity` database schema for the backend, so I could start populating the content database.

Here is an example of the `teamMember` component, which holds information for each team member.

```
export default {
  name: 'teamMember',
  title: 'Team Members',
  type: 'document',
  fields: [
    {
      name: 'name',
      title: 'Name',
      type: 'string',
      validation: (rule) => rule.required(),
    },
    {
      name: 'job',
      title: 'Job Title',
      type: 'string',
      validation: (rule) => rule.required(),
    },
    {
      name: 'image',
      title: 'Image',
      type: 'image',
      description: 'Ensure this has a transparent background',
      validation: (rule) => rule.required(),
    },
  ],
  preview: {
    select: {
      title: 'name',
      media: 'image',
    },
    prepare({ title, media }) {
      return {
        title: `${title}`,
        media,
      }
    },
  },
}
```

The code above creates an `object` in the schema under the name `teamMember`, which has three fields: `name`, `job` and `image`. It also creates an object preview with the team member's name and image, for a better user experience when inputting content.

This is how it shows in the CMS:

[IMAGE REDACTED]

When clicking on each member, an editor opens up where we can edit the member properties:

[IMAGE REDACTED]

Database content modelling

The journey to building a content model with Sanity was an important part of the project. The nature of data here implies some relationships between different pieces of information. I made sure to trace the more obvious connections first, so I could build the core of the schema.

As an example, each project page has the avatars of the team members who work on it, like so:

[IMAGE REDACTED]

This is possible since, inside the `project` object in the Sanity schema, I have added a reference to the `teamMember`:

These are the fields in the CMS:

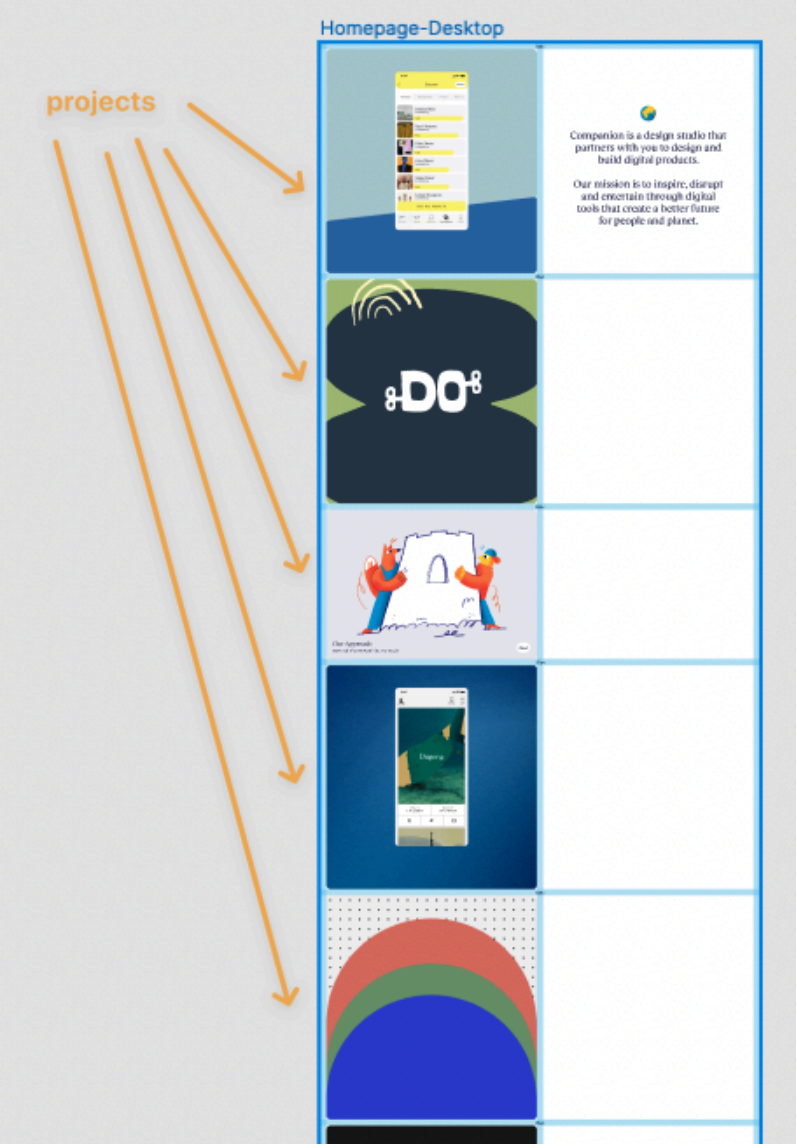
[IMAGE REDACTED]

This is a very important concept here. The structure of the content is very close to the relational database paradigm since data is grouped in a logical way and is getting interconnected with other data from other groups. Instead of `tables`, we have different types of `objects`, which form `arrays of objects` or/and `documents`. All of these can refer to one another, which ultimately creates a graph of structured data, which can be queried.

Components reusability

One of the main principles in designing and implementing this project is the concept of components. Blocks that are either identical visually or in terms of purpose are grouped together and we try to serve them through one piece of code.

One example is the `projects` section.

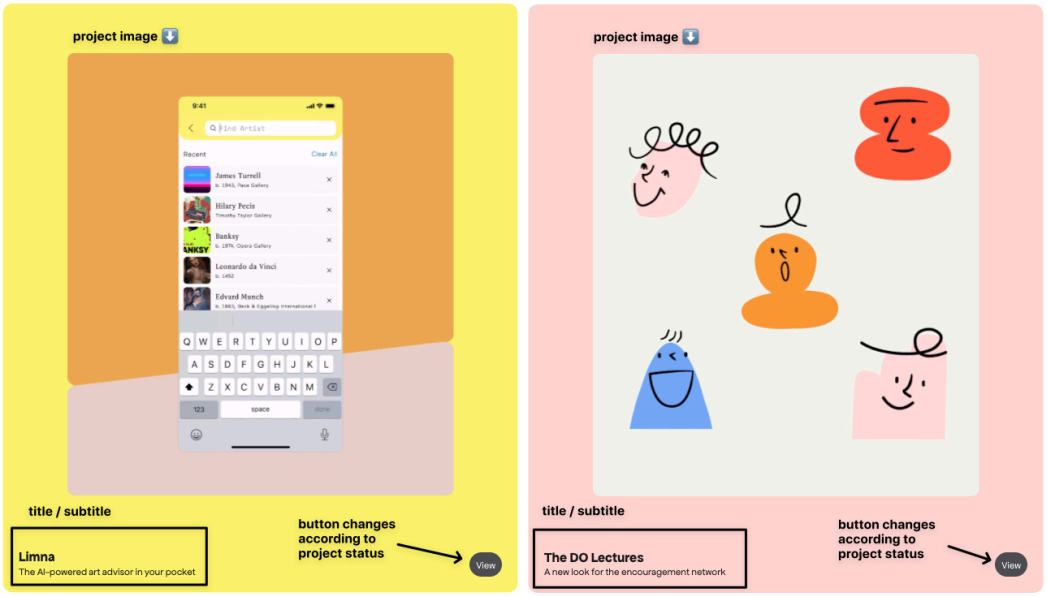


All these different sections here are served by the same piece of code in the front and back end.

The thumbnail cards are being rendered by the `CardHome.tsx` component, which accepts the following as props: `media`, `meta`, `subtitle`, `theme`, `title`, `cardButtonLabel`, `status`, `style` and `backgroundColor`. This data comes from the CMS since each project has its dedicated properties object.

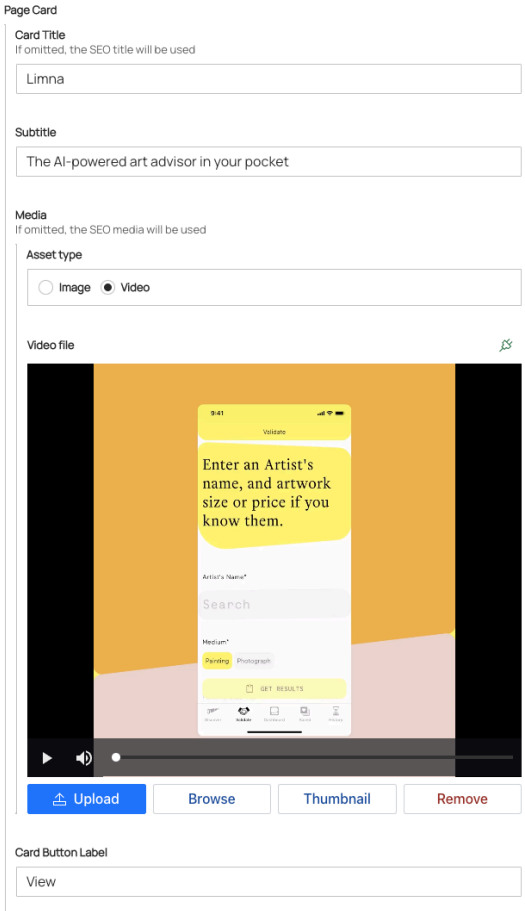
Once this data is available to the component, the project card renders with the project image, background colour, title project and all other properties mentioned above.

Here is an example of two project cards side by side - we can see how these common properties are applied:



These are the sections in the CMS that send this data to the front end.

The same process has been followed in all pieces that follow the same patterns of repetition. In this way, we can freely create more content that comes from much less code. Most bugs can be identified in a more efficient way and the codebase is easier to maintain.



[CLIENT FOUR]

About the project

This website was built upon request from [CLIENT FOUR]. According to the company, **[CLIENT FOUR]** uses the power of marketing to communicate the climate projects that will save our planet. Their goal is to make scientific information more accessible for environmental start-ups and help them be heard and establish their message.

Planning & Design

During the discovery phase of the project, I had the chance to explore the idea behind [CLIENT FOUR], its position in the industry and its values. [COMPANY NAME] arranged a workshop, where we could delve deep into the drives, needs and aims of the client.

The main consideration

There is a gap between the amount of scientific data and how it translates to the audiences and they stressed how important it is for them to bridge this gap. One of the main considerations was to make the site look 'scientific' enough so it speaks to the academic audience, but at the same time accessible, so it could attract people regardless of background. "Green" and "spacious" were two main keywords that kept coming up in discussions.

Time restrictions

The timeframe was quite tight for this project, based on the client's scheduling. As they wanted to launch the soonest possible, the agreed time ended up being 4 weeks, including QA testing and potential amends the client could request.
There was a discussion on where the site content would be stored - would it be static or dynamic? The client said they would like to be able to maybe change the content in the future, which meant I would have to implement some CMS for the backend, such as `Sanity` for example. Due to the time constraints though, as well as the higher cost of this, a static site was decided, so all data is hardcoded.

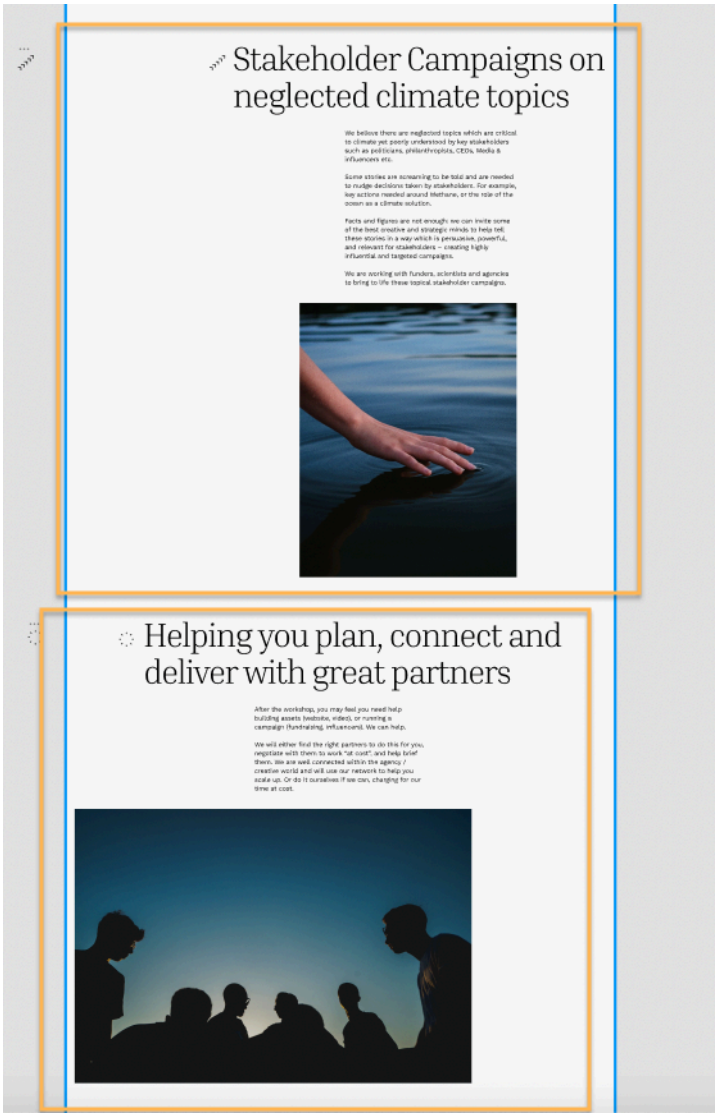
Using components with `Storybook`

What is `Storybook`?

For this project, I am using `Storybook`, a tool for building UI components in isolation. This is particularly helpful given the nature of the project. The website has more of an editorial feeling and everything is arranged visually into blocks.

[IMAGE REDACTED]

First, I identified layout similarities among all these blocks and grouped them together. The following two blocks seem like they could be grouped together. They both have an svg icon, title, some content and an image, which translates into the props `({icon, title, content, image})`, but present variations in their layout.
It only makes more sense for these 2 to be coming from the same component. I have added an extra prop `pageLayout`, which controls which of the two layouts will be activated by the block calling it.

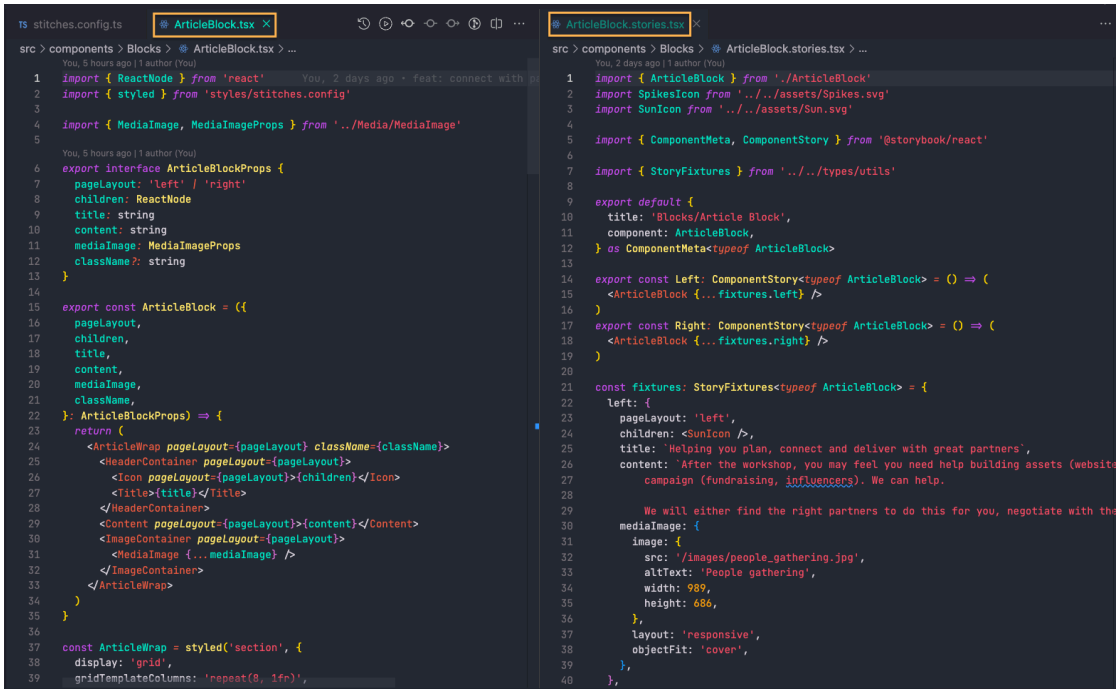


This is how my code is organised. I have an `ArticleBlock.tsx` file, which is the component itself, as well as an `ArticleBlock.stories.tsx`, which controls the component's entry in `Storybook`.

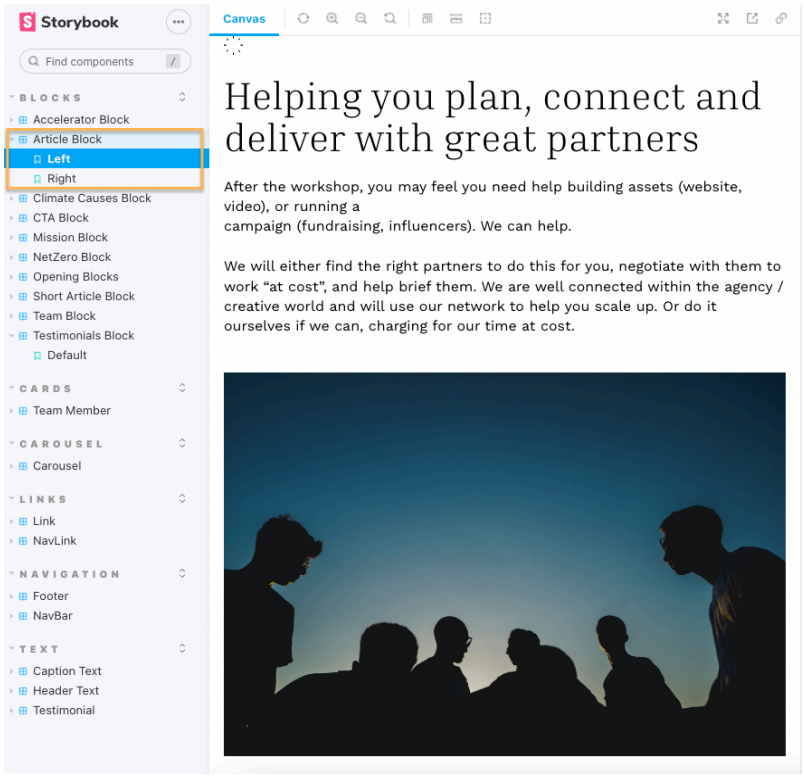
- In the `stories` file, I am creating different versions of the component, by passing different fixture data based on the design.
- Here we have 2 variations, `left` and `right`, that are being passed like this into storybook:

```
const fixtures: StoryFixtures<typeof ArticleBlock> = {
  left: {
    pageLayout: 'left',
    children: <SunIcon />,
    title: `Helping you plan, connect and deliver with great partners`,
    content: `After the workshop, you may feel you need help building assets (website, video), or
running a
        campaign (fundraising, influencers). We can help.

        We will either find the right partners to do this for you, negotiate with them to work "at
cost", and help brief them. We are well connected within the agency / creative world and will use our
network to help you scale up. Or do it ourselves if we can, charging for our time at cost.`,
    mediaImage: {
      image: {
        src: '/images/people_gathering.jpg',
        altText: 'People gathering',
        width: 989,
        height: 686,
      },
      layout: 'responsive',
      objectFit: 'cover',
    },
  },
  right: {
    pageLayout: 'right',
    children: <SpikesIcon />,
    title: `Stakeholder Campaigns on neglected climate topics`,
    content: `We believe there are neglected topics which are critical to climate yet poorly understood
by key stakeholders such as politicians, philanthropists, CEOs, Media & influencers etc.

    Some stories are screaming to be told and are needed to nudge decisions taken by stakeholders. For
example, key actions needed around Methane, or the role of the ocean as a climate solution.

    Facts and figures are not enough: we can invite some of the best creative and strategic minds to
help tell these stories in a way which is persuasive, powerful, and relevant for stakeholders - creating
highly influential and targeted campaigns.
```

```
    We are working with founders, scientists and agencies to bring to life these topical stakeholder
campaigns.`,
    mediaImage: {
      image: {
        src: '/images/ripples.jpg',
        altText: 'Ripples in water',
        width: 541,
        height: 686,
      },
      layout: 'responsive',
      objectFit: 'cover',
    },
  },
}
```



This is how this information appears in Storybook 🚀



I followed the same pattern of work for the whole project. Blocks that accept the same types of arguments are stemming from a common component, which is being reused per case. This way I am promoting reusability, while at the same time making the code more sustainable.

Procedures & version control

The way the request was given by the client during our workshops, helped us understand our main workflows.

- the user needs to land on a homepage, which should provide all necessary information regarding the agency and its principles.
- the user needs to know more specifics about the agency’s actions and projects.
- the user needs to have access to a privacy policy, so they understand how CA utilises their data
- the user needs to be provided with a 404 page, in case their request cannot be fulfilled

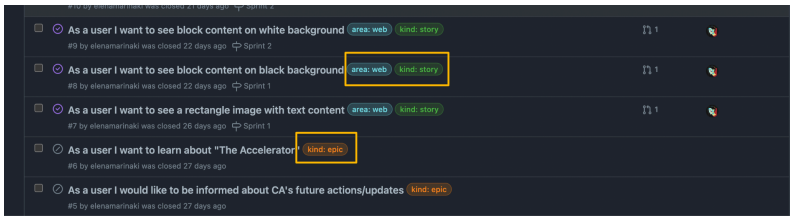
Once the design team had agreed on the branding and assets with the client, they created the first iteration of how the website will look like. Although not all things

were finalised yet, the development team had to start working on setting up the project and creating the components that were implied by the design.

The reason we could do that is that the suggested layout was unlikely to change, so since we use `Storybook`, we could start creating our library of components there and use them as the client pleases, with any assets they provide.

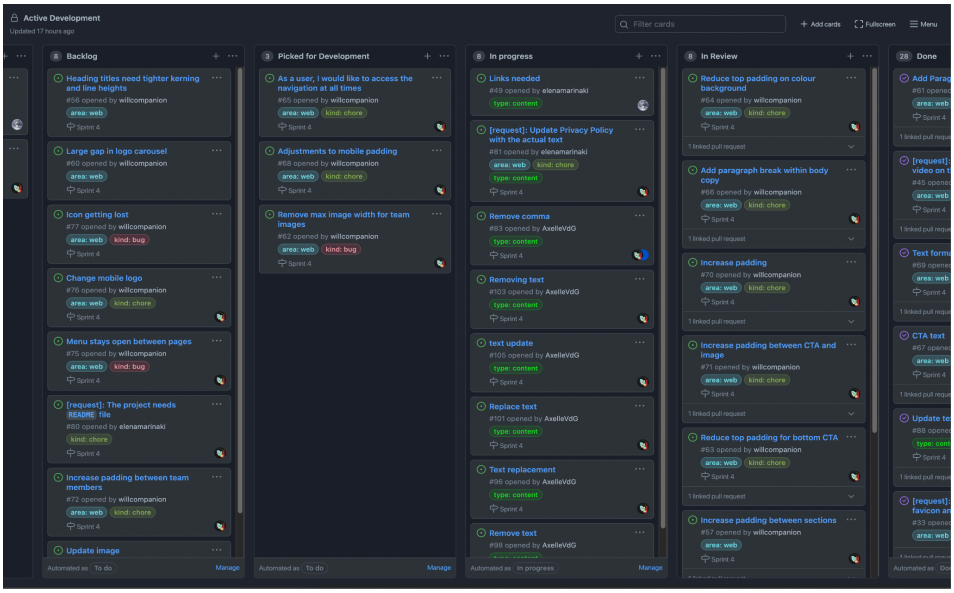
We use `GitHub` as a version control system and tool for communication among developers.

- The 4 points above served as our `EPICS` in developing the application. These are generic tickets that only get fulfilled by many sub-tickets and come together at the end of the building period when I put all pages together.
- this is part of the `Issues` page - all tickets are tagged, and `stories` derive from the bigger `epics`



- every issue is tagged with very specific information
    1. the developer who's working on it
    2. the suggested area (web, cms, content, story, epic, styling, bug, etc..)
    3. the project board
    4. the sprint number
    5. any linked PRs.

As soon as I would start working on a ticket, I would tag it as `in progress` and it would move under the same category in our automated `kanban` board on GitHub.



The board is the main endpoint of reference for the whole team. Everybody could see in which stage of the build I was at times, the open tickets, what was coming next etc. This helped communication a lot by reducing unnecessary info and focusing on the things I actually had to implement.

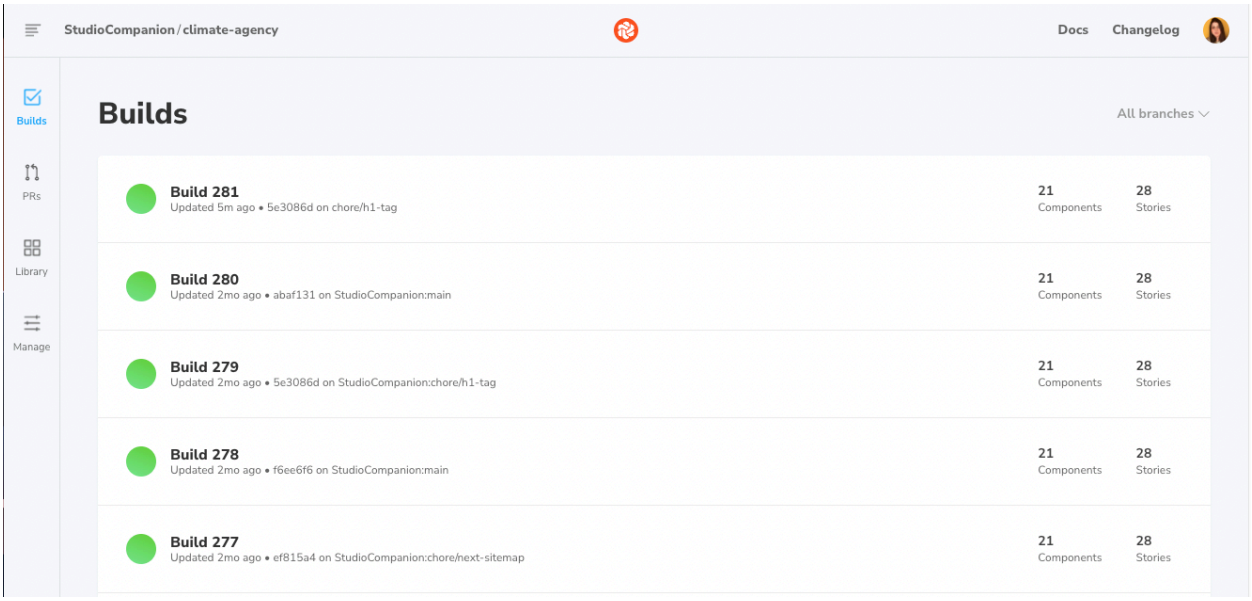
This workflow proved especially helpful during the QA, as all team members could write requests under `Issues`, which I then sorted and started working on. In every new PR, I would link the issue back and tag the member who would need to do the reviewing and approving.

Continuous Integration

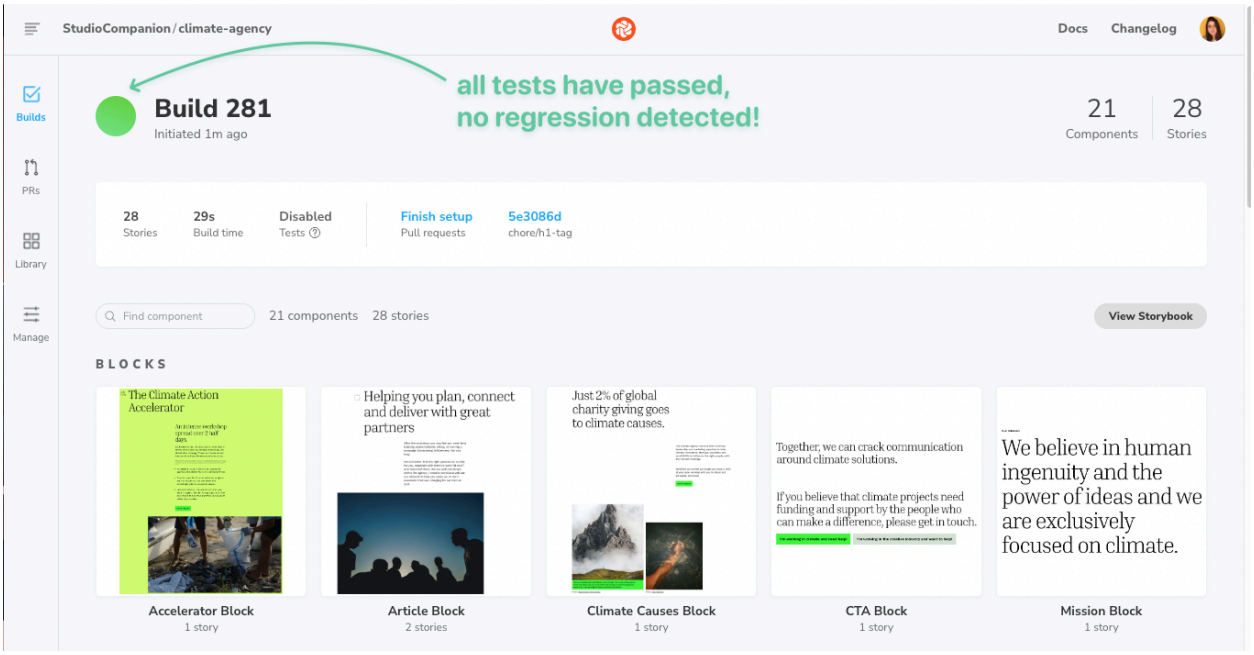
The automated tests provided by `Vercel` (the deployment service) as well as the `Chromatic` (for `Storybook`) were running every time I pushed a new PR. This way I could ensure that every new piece of code I was adding was not breaking the existing one and that all components kept working as expected.

I set the workflows for `Chromatic` under the `.github` > `workflows` folder. I ask for the components to be redeployed there with every new PR so that `Chromatic` can perform its **visual regression testing**.

This is a screenshot from `Chromatic`. It shows all successful builds of my `Storybook` components, as indicated by the green coloured circle:



Opening the last build, I can see all current components of the website along with their stories. Since this process is happening automatically through CI, I do not have to inspect `Chromatic` unless a test fails during deployment.



This setup provides the necessary flexibility on projects with lots of components like this one and makes the QA process quite smoother, not only for me but for the team as well. When a change that alerts `Chromatic` happens, I can inspect it and decide if it comes from regression, in which case I fix it. If not, that means it comes from an updated component and I simply replace it with the newer version by accepting the changes.

Where is the content of the website stored?

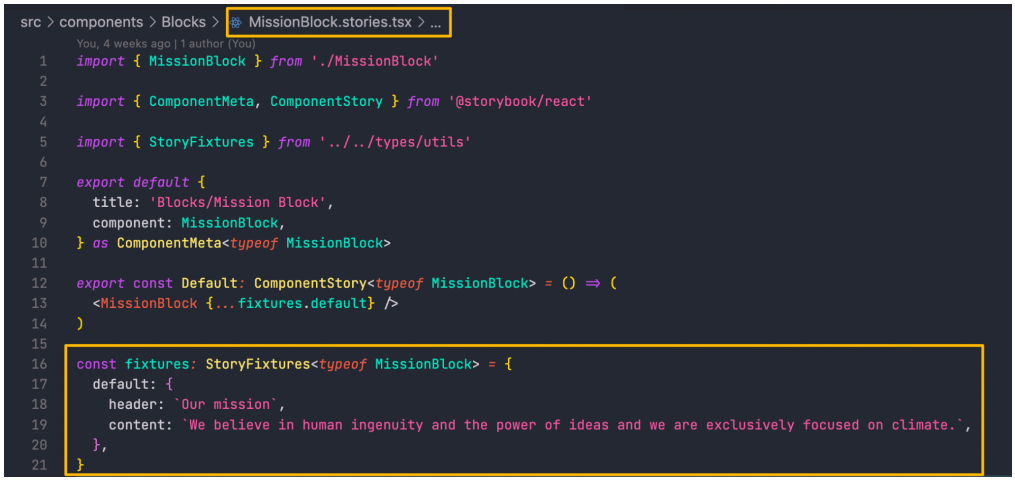
This is a static website, where information is presented in an editorial style. Due to this nature, we opted out of using a CMS, since this data is not going to be changing often. Instead, all data/content can be found inside the package.

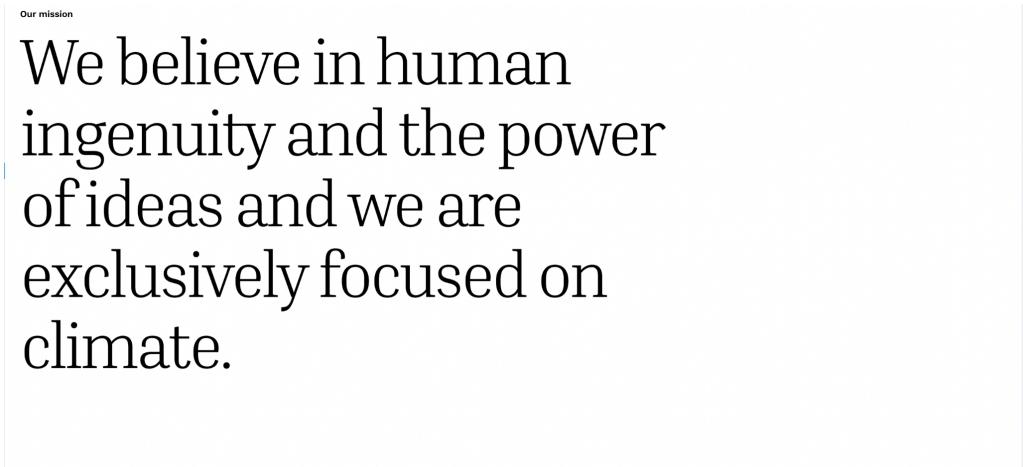
Different outputs, different sources 🔀

1. Data in `Storybook`

Since we are using `storybook`, the content we pass in the components and which can be accessed through the `storybook` server for our stories comes from fixtures, which are placed inside each `.stories.tsx` file.

In the following example, the `MissionBlock` has its own `MissionBlock.stories.tsx` file, in which we pass the content, which we will see in its story.





2. Data in Pages

The content we see when we visit the website though comes from a different source. For this, we have a dedicated folder under `src` called `data`, in which we store all content in objects, which are being passed as `props` into their corresponding components.

Let's take the `Background` block as an example this time - this is how we store the data object in our `homepage.tsx` file:

```
export const ourBackground = {
  textPosition: 'left',
  header: `Our background`,
  content: `We are a team of experts in marketing and communications, with prestigious careers leading global consumer brands
  and charities.`,
  imageSection: {
    mediaImage: {
      image: {
        src: '/images/Mad-Don.jpeg',
        altText: 'Mad Don',
        width: 343,
        height: 348,
      },
      layout: 'responsive',
      objectFit: 'cover',
    },
    caption: `Inspired teamwork can achieve the impossible. Here "Mad Don", Donal Arabian, Chief of the Apollo Test Division.
    Credit NASA Archives.`,
  },
} as ShortArticleBlockProps
```

And this is how this data is being passed into the component calling it (along with the declaration of its types, since we are using `Typescript`):

```
export interface ShortArticleBlockProps {
  textPosition: 'left' | 'right'
  header: string
  content: string
  link?: LinkProps
  imageSection: {
    mediaImage: MediaImageProps
    caption: string
  }
  className?: string
}

export const ShortArticleBlock = ({
  textPosition,
  header,
  content,
  link,
  imageSection,
  className,
}: ShortArticleBlockProps) => {
```

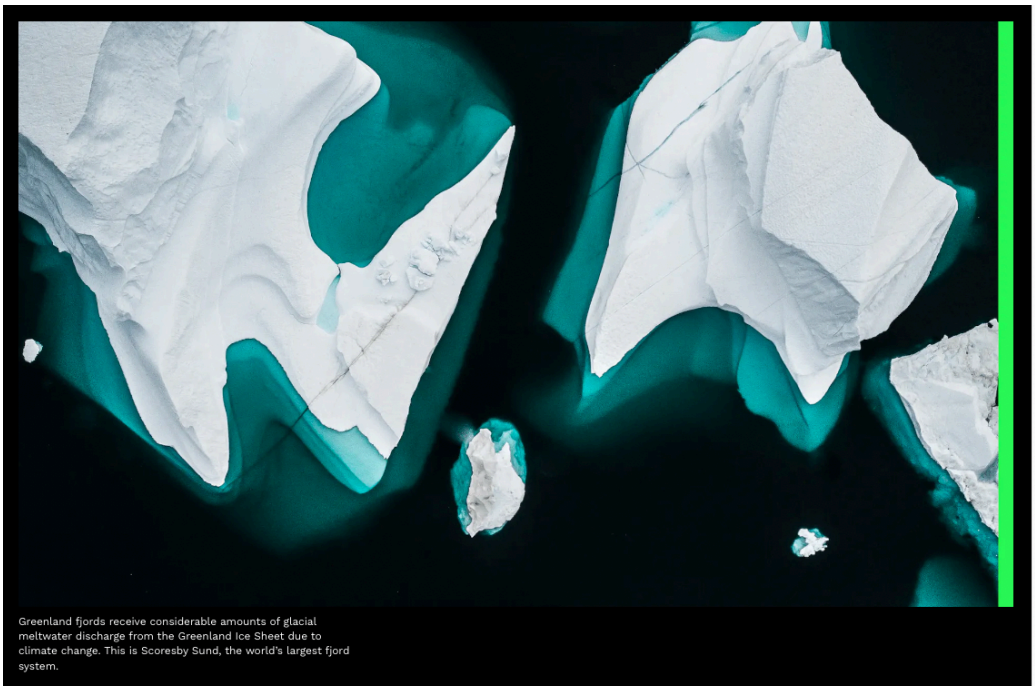
Requested changes from the client after release 🌍

As it is very often the case, the client had recently had a couple of requests, some additional changes/features they'd like, a couple of months after the initial release.

[IMAGE REDACTED - shows an internal team conversation]

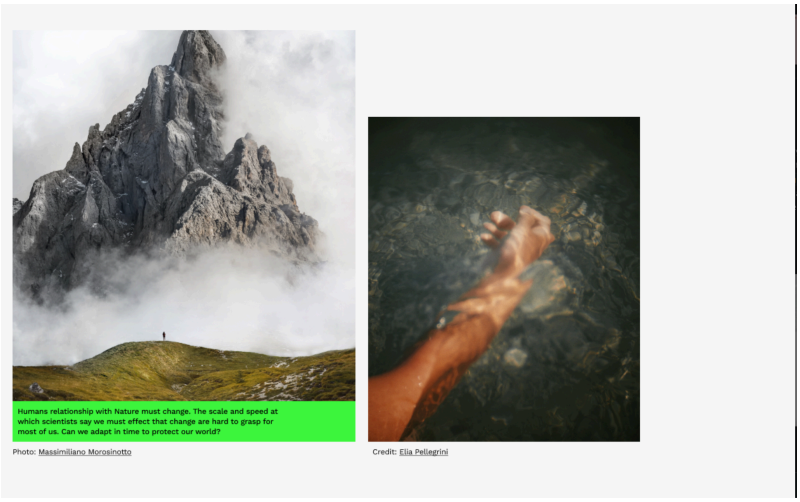
The initial request included mostly copy changes, but later on, a couple more things were added, since one of the board members of the client's company pointed out that the image captions were not very "capturing" and were most probably skipped by most users.

This is how it looked in the first release:



Our producer, [PRODUCER / DESIGNER], informed us of the client's concern, and our senior designer started working on some amends, so I could have some designs to work with.

Will's proposed solution was to place the caption inside the green strip, which is already part of the brand of [CLIENT FOUR]. This way, we could have something like the image below: Images can have a caption (green strip), credit, both or none.



The client approved this solution and so I proceeded to implement it.

Technical considerations regarding the captions

In the initial implementation, the green strip was part of the rendered image. I chose this solution at the time since the time was extremely limited and we had the images already in our design files.

With the new design though, this wouldn't work, so I let the team know, so they can prepare new image files to include and also that I have enough time to work on it based on my estimation.

[IMAGE REDACTED - shows an internal team conversation on Slack]

This time I made a separate component for the green strip, calling it `InnerTextCaption` with an `absolute` position property. Any image container now assumes a `relative` position, so that the green part can always stay at the bottom of each image and be responsive along with the image.

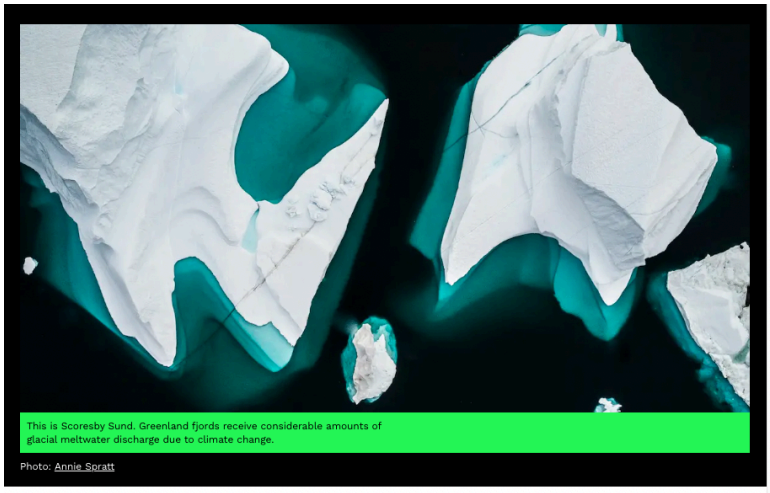
This is how the `InnerTextCaption` component works:

```
import { styled } from 'styles/stitches.config'

interface InnerTextCaptionProps {
  children?: string
  className?: string
}

export const InnerTextCaption = ({
  children,
  className,
```

```
}: InnerTextCaptionProps) => {
  return (
    <InnerCaptionContainer className={className}>
      <p>{children}</p>
    </InnerCaptionContainer>
  )}

const InnerCaptionContainer = styled('figcaption', {
  position: 'absolute',
  width: '100%',
  backgroundColor: '$green',
  color: '$black',
  left: 0,
  bottom: 0,
  p: '$8',
  fontFamily: '$workSans',
  fontWeight: '$medium',
  fontSize: '$XXS',
  lineHeight: '$XXS',

  letterSpacing: '$small',
  whiteSpace: 'pre-line',

  '& > p': {
    maxWidth: '430px',
  },
})
```

Now all captions (if existing) are rendered through this component (`InnerTextCaption`), whereas the image credit (if existing) is rendered through the already existing `TextCaption` component (that was initially rendering the caption).

An example of the result is this:



Now the component is easily reusable and this way I was able to make all changes in a fast, efficient and error-free way. It is also easy to spot potential errors since the component has its own entry in `Storybook`, which runs `Chromatic` for visual regression every time I push a new commit through GitHub.

The carbon footprint of the website

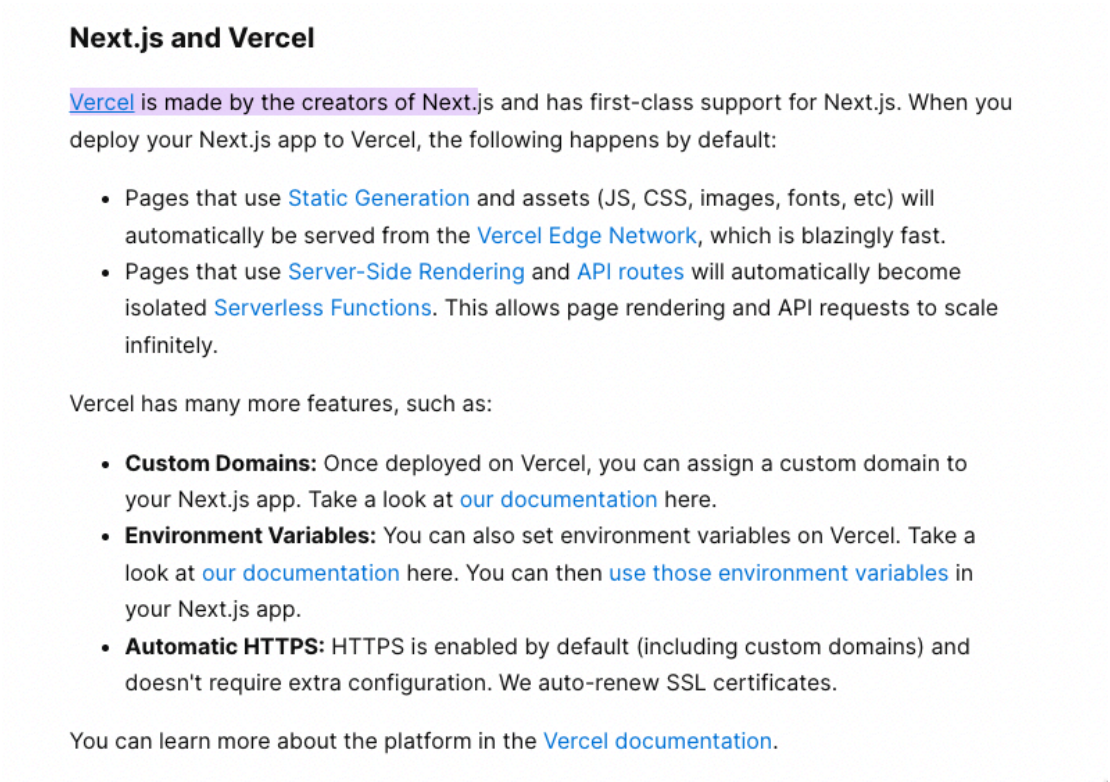
One important consideration (as it is for all projects) is the carbon footprint of the website, even more so due to its nature being a climate website.

The website is live at [WEBSITE URL]. When I ran this URL at [WEBSITE URL] which assesses how carbon footprint friendly the project is, I got this result.

I made an actual effort to reduce the carbon footprint as much as possible by following these actions:

- creating as many reusable components as possible. This helped in reducing the size of the code, as well as unnecessary repetition and clutter. Things that look the same or almost the same are rendered by the same component.
- including only necessary assets for the website. I also asked the designer team to provide compressed versions of the images, so this way I could significantly reduce the overall size of the build.
- building the media components using the `next/image` API, which provides an optimal way of handling image files, in terms of quality, sizes, lazy-loading etc.

As you see from this result, one thing that could be subject to improvement is the site hosting itself. The website is currently hosted on `Vercel`, which is a common (and obvious) choice for projects built with `Next.js`, since according to their website:



Next.js and Vercel

Vercel is made by the creators of Next.js and has first-class support for Next.js. When you deploy your Next.js app to Vercel, the following happens by default:

- Pages that use Static Generation and assets (JS, CSS, images, fonts, etc) will automatically be served from the Vercel Edge Network, which is blazingly fast.
- Pages that use Server-Side Rendering and API routes will automatically become isolated Serverless Functions. This allows page rendering and API requests to scale infinitely.

Vercel has many more features, such as:

- **Custom Domains:** Once deployed on Vercel, you can assign a custom domain to your Next.js app. Take a look at our documentation here.
- **Environment Variables:** You can also set environment variables on Vercel. Take a look at our documentation here. You can then use those environment variables in your Next.js app.
- **Automatic HTTPS:** HTTPS is enabled by default (including custom domains) and doesn't require extra configuration. We auto-renew SSL certificates.

You can learn more about the platform in the Vercel documentation.

My company has a `Vercel` account, where we are hosting our `Next.js` projects since this has all the benefits seen above and provides a seamless experience for both users and developers. This is a trade-off we make as a company, which I am sure we might be able to re-assess in the future with all available choices we currently have in the market.

[CLIENT ONE]

About the project

[CLIENT ONE] is a revolutionary enterprise Saas/Paas customer analytics platform that knows when and why a customer will buy from one retailer/brand instead of another.

“The [CLIENT ONE] platform was created in response to the desperate need for better customer analytics, capable of looking outward from a business across a wider sector. Traditional customer analytics tend to be predominantly inward-looking; they only really give detail on the company commissioning them.

The [CLIENT ONE] platform defines a value for 'relative attractiveness' (RA) for a given experience and customer mission. Growth comes from increasing RA, understanding how to increase RA is what our friction-versus-reward tools do.”

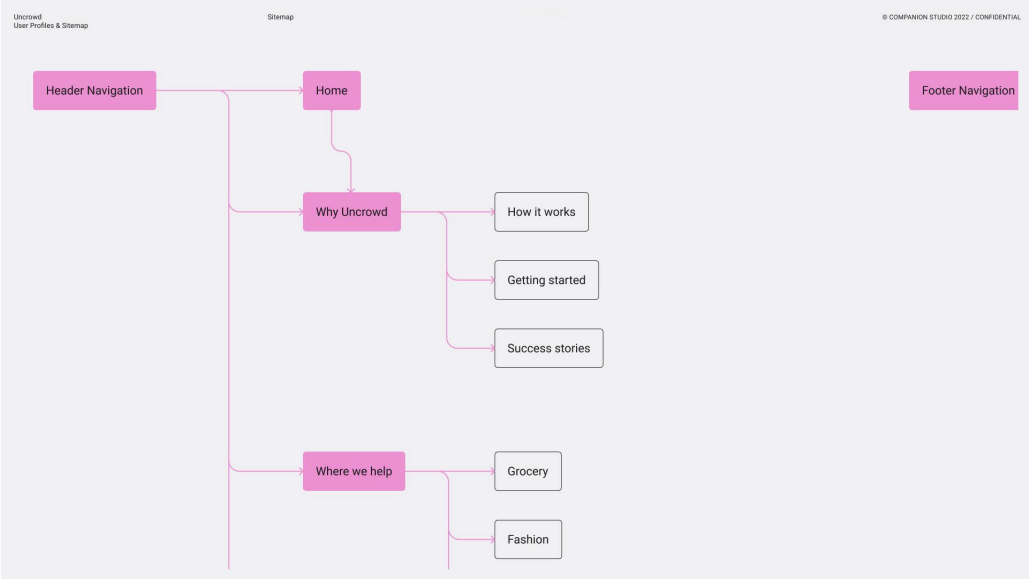
[Image shows brief provided by client one]

From the company’s brief

What they wanted to achieve in partnering with [COMPANY NAME]

The previous website was built more than three years ago. With their increasingly complicated product offering, the people at [CLIENT ONE] found it was no longer sufficiently informing and engaging the enterprise’s audience about their product and what they do.

Their main goal was to improve how they communicate their product and use fresh visual language to talk about their identity. In terms of the main actions that somebody could take on the website, this is one of the flowcharts we ended up with during the discovery phase:



Examples & code evidence

The ‘Quiz’ pages

One of my main contributions to the website was the implementation of the `Quiz` pages. This is essentially a large form, broken down into smaller steps, in which the potential client can discover what are their needs and ultimately communicate with the enterprise.

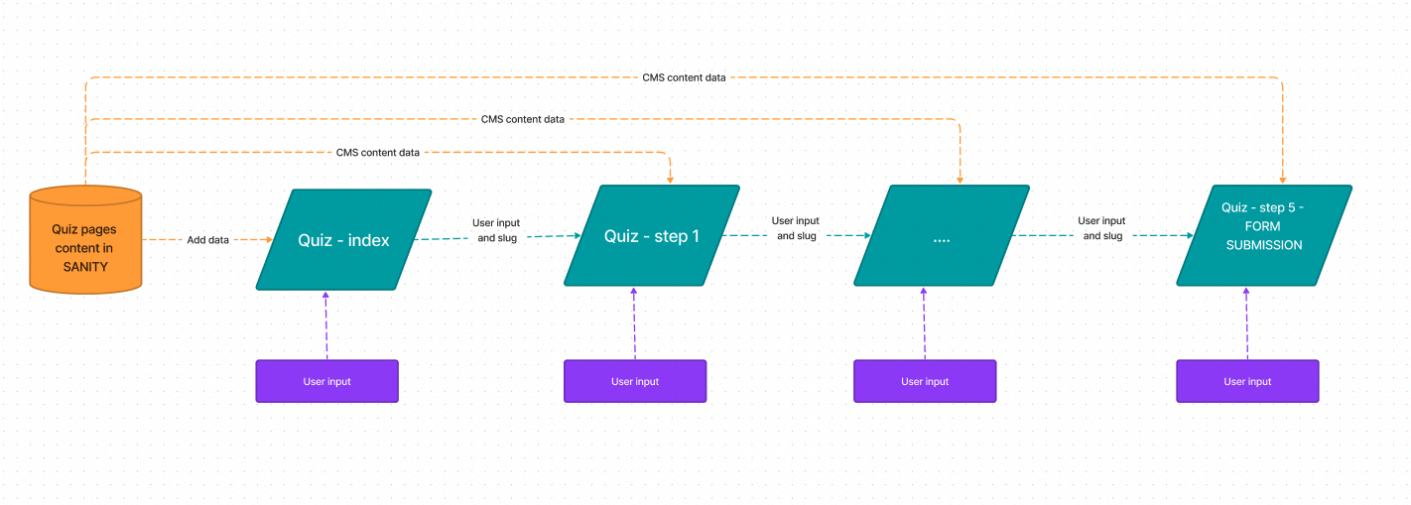
The user can launch the quiz by clicking the `Get started` button from the Homepage:

[Gif shows client page and interaction with the ‘get started’ button]

How this works

Through a series of steps, the user is prompted to give answers to the provided questions. Every answer is saved and used in the subsequent steps till the end.

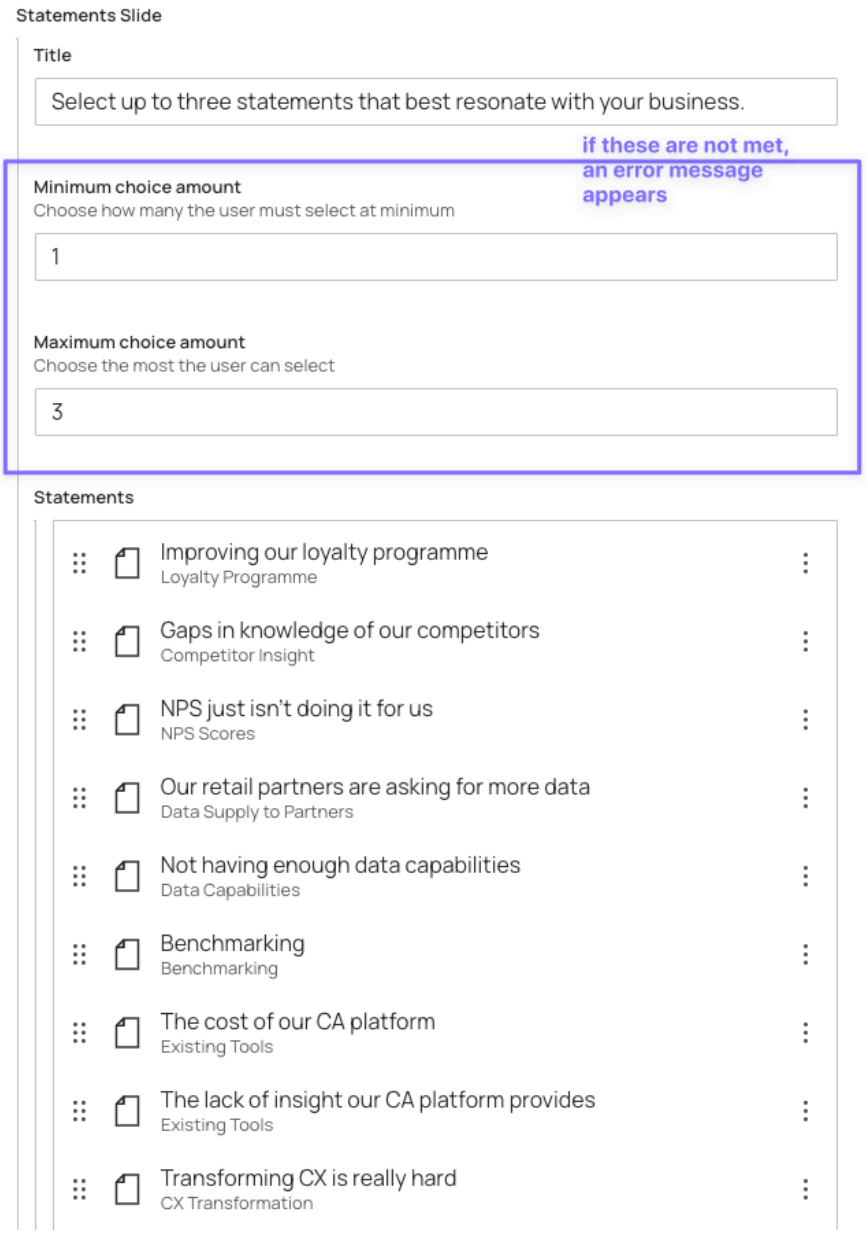
The whole quiz is a page which links to five different steps through dynamic routes. What this practically means is that for each potential route there is a specific layout and functionality, but its main content depends on the data the user has inputted, so it renders accordingly.



This flowchart summarizes the functionality:

Each one of the steps renders some static content that comes from `Sanity`. For example, see how `step 1` looks on the website and compare it with its `Sanity` entry:

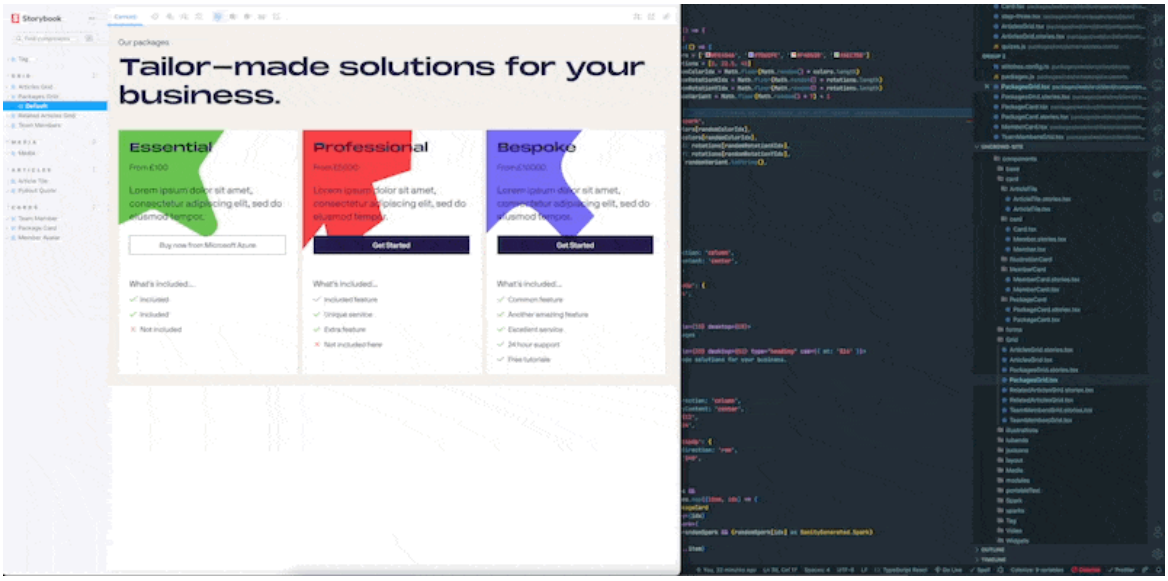
Each This is the entry for `step 1` in the `CMS`. All the content the user access and choose from through this page comes from here:



Using `svg` as visual language & technical considerations

One of the elements decided as part of the brand was the `spark`. The spark appears everywhere in one form or another in a range of rotations and in two versions: filled or outlined.

They are used in various components and are always displayed using a variation of the X & Y axis (and never flat).



Client communication

The following component takes as props a card component that represents a company package available for sale. We can have up to three cards that state the features of each package and the differences compared to the others.

Initially, there were different considerations for the styling of the `package card`. The client wanted to use some colour in a way that is representative of the brand (playful, intriguing) but at the same time not overflow the component with more information. We ended up with this particular solution, according to which a spark sits at the top left of each card, with each colour, variation and rotation randomised each time.

The client asked for having potentially the ability to choose the spark configuration through the CMS, as this is the case for other parts of the application as well, but we opted out of that, since:

- it is a smaller component, reused in different places in the website, and selecting a spark for each one of these cases would be a tedious process for the content manager
- with the chosen solution we bring some extra motion, making these parts visually dynamic and more playful

Technical considerations

The state that “decides” the random properties for the sparks is being handled by the parent component. This is in accordance with our practice to include functionality in children's components as little as possible and only when absolutely needed.

I return as many state objects as the number of cards of available packages. Each state represents a `spark` object with all the information that needs to be rendered.

```
const [randomSpark, setRandomSpark] = useState(
    packages?.map(() => {
      return {
        _type: 'spark',
        fill: '',
        stroke: '',
        rotationX: 0,
        rotationY: 0,
        variant: '1',
      }
    })
  )
```

All these properties change every time the component re-renders, that is on a “refresh” in this case. To make this happen, I had to randomise these properties inside a `useLayoutEffect` for as many sparks as we have.

```
useLayoutEffect(() => {
    setRandomSpark(
      packages?.map(() => {
        const colors = ['#FE4546', '#7D6EFE', '#F4D53B', '#6EC750']
        const rotations = [0, 22.5, 45]
        const randomColorIdx = Math.floor(Math.random() * colors.length)
        const randomRotationXIdx = Math.floor(Math.random() *
rotations.length)
        const randomRotationYIdx = Math.floor(Math.random() *
rotations.length)
        const randomVariant = Math.floor(Math.random() * 7) + 1

        return {
          _type: 'spark',
          fill: colors[randomColorIdx],
```

```
          stroke: colors[randomColorIdx],
          rotationX: rotations[randomRotationXIdx],
          rotationY: rotations[randomRotationYIdx],
          variant: randomVariant.toString(),
        }
      })
    )
  }, [packages])
```

Why `useLayoutEffect`?

I had to use `useLayoutEffect` instead of the usual `useEffect` in this case. When I first tried `useEffect`, although the sparks were rendering correctly they would never change on “refresh”. It seemed like no more new randomised values were produced, which was unexpected.

After some searching, I had to ask for help from the senior developer of the team and discuss what the possible problem might be in this case. He suggested that although my thinking was correct to place the randomisation inside a `useEffect`, I was forgetting something important: the `spark` is an svg and every time some or all of its properties change, another svg has to render, which is a change in the DOM.

(I had to go a bit deeper and do some reading, this article was particularly helpful in understanding this nuanced difference between the two hooks.)

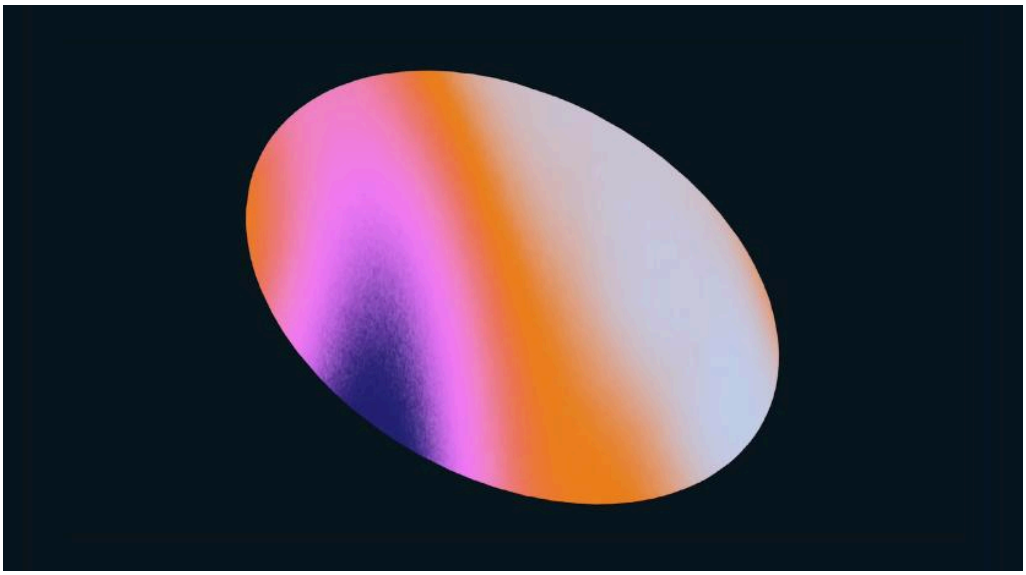
For this and other similar cases (non-static assets), `useLayoutEffect` works best, since this hook runs synchronously immediately after React has performed all DOM mutations.

Lastly, I pass the state information into the `spark` prop of each package card like this, making sure I let the component know the type of the spark is the same as the one coming from the CMS configuration (since this is how we have typed the `spark` in the whole project and this is how it is being recognised by the components):

```
{packages &&
  packages.map((item, idx) => (
    <PackageCard
      key={idx}
```

```
      spark={
        randomSpark && (randomSpark[idx] as SanityGenerated.Spark)
      }
      {...item}
    />
  ))}
```

[CLIENT THREE]



About the project

[CLIENT THREE] is a talent consultancy for hypergrowth enterprises based in California. They focus on building brands, acquiring talent and sharing their hiring expertise with entrepreneurs and tech companies.

This project is quite extensive in terms of technologies used in the frontend, due to its complicated interface and advanced interactions. A great part of the things that happen visually are built with the `Three.js` library and mostly in `Vanilla Javascript`.

My work on this project was explicitly focused on building components in the front and back end and styling them according to designs. I have used the usual stack of `Next.js`, `Sanity CMS` and `Tailwind`.

Examples & code evidence

Integrating `Iubenda` and cookie banner

The website is using cookies and presents the user with an informative banner when they land on the page for the first time.

[Image shows company site]

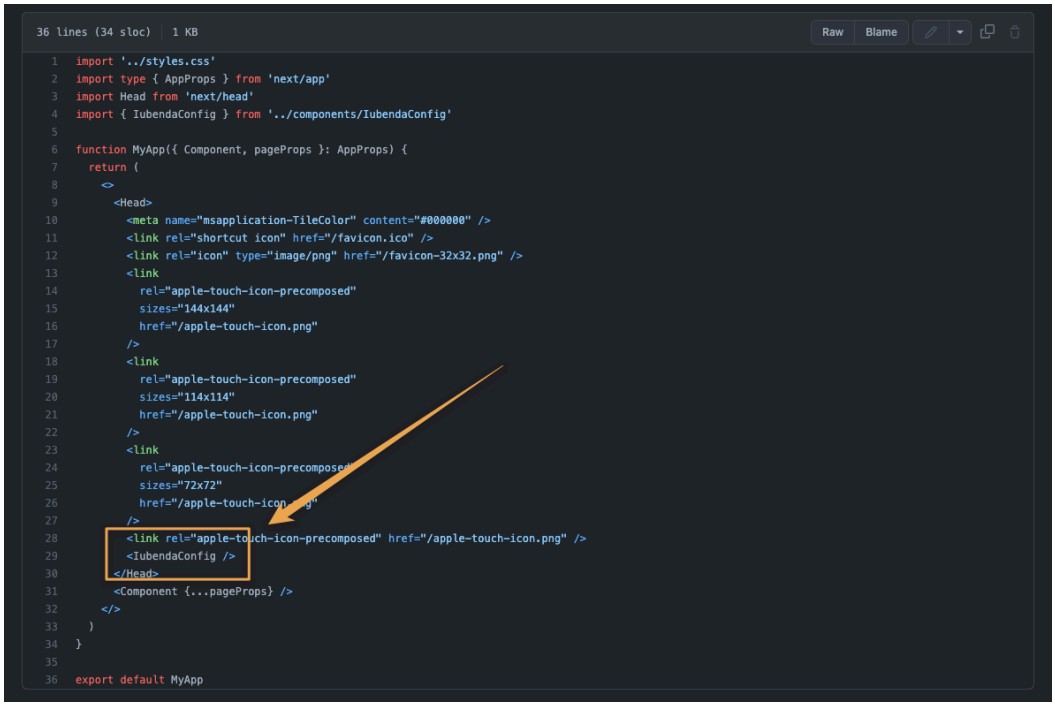
As with all websites of similar services, this one too complies with European and international laws regarding user privacy and personal data handling.

To manage the legal requirements and make sure they are covered properly, we use an external service called `Iubenda`, which provides businesses with all necessary solutions to ensure their websites or apps are compliant with the law across multiple countries and legislations.

Here, I set up the company's page on the Iubenda's website and included a cookie solution along with the Privacy and Cookie Policy. There are quite a few services available, but here I have chosen the two main ones.

[Image shows cookie options associated with company url]

The service generates a script for each solution, which I pass into our codebase. For cookies, I pass the code into a regular `tsx` configuration file with the `dangerouslySetInnerHTML` method. Then I just import it in the root of the project, in the `_app.tsx` file under the `<Head />` tag, like so:



What remains to be done after this process is the styling of the banner. This can get a bit tricky sometimes since the code that comes from `Iubenda` is in markup with pre-defined classes attached, which cannot be changed. So, to apply a specific style, I usually have to track down the class name of a specific element via `devTools` in chrome, and then use it with the CSS framework I am working with to apply the design.

The styling rules should be part of the global css file, so here I have added everything in the `../styles.css` that is being called in `_app.tsx`.