Activity type 1:

Wireframes and prototypes

For my internship project, I was tasked by my manager with redesigning and modernizing the company's e-commerce platform. The existing site was built using WordPress, which resulted in slow performance, with page load times often taking tens of seconds, and occasional stalling. Additionally, my manager expressed concerns about the high maintenance costs associated with the platform. To address these issues, I was also asked to transition the e-shop to a more efficient and cost-effective technology stack.

After discussing the project requirements with my manager, I proposed adopting a technology stack consisting of Directus for the backend and Next.js for the frontend. Having previously worked on several projects using Next.js and being aware of the positive reviews regarding Directus, I was confident that this combination would address the performance and maintenance cost concerns. Upon evaluating my proposal, we decided to proceed with this technological approach.

My manager emphasized the importance of creating an intuitive design for user navigation, as well as a user-friendly interface for administrative tasks. Directus provides a highly straightforward and efficient management system for the database and the data stored within it. After demonstrating an example of Directus' capabilities, my manager was convinced that it would be an excellent fit for our project requirements.

1. Wireframes

Upon agreeing on the desired technology stack, I promptly began designing the modernized version of the website. The first step in this process was to create wireframes, which serve as a foundational guide for the subsequent development and design of the prototypes. Given that modern design principles prioritize a mobile-first approach, I initiated the wireframe creation process with the mobile view (small screens). For this task, I utilized the platform Draw.io, a tool with which I am well-acquainted.

Wireframe of the homepage for mobiles:

In this wireframe, we observe a clear separation of the view into three distinct sections: the top/header section, the main/content section, and the bottom/footer section.

The header section is designed to inform visitors and customers about the available links through the menu, any additional information specified by the manager, and the categories of the e-shop.

The main section features two carousels of products: one for suggested products and another for newly added items to the shop.

The footer section functions as a submenu, providing additional links to legally related pages, as well as information regarding payment methods and the site's copyright.

Both the header and footer sections remain consistent across the entire website, with minor adjustments made to the header section to accommodate the screen size and the content displayed on each page.

Wireframe of the home page for desktop view:

Similar to the mobile view design, the desktop view is also divided into three sections: header, main, and footer. The primary difference in the desktop design is the size of the images. Since larger screens can accommodate larger graphics, the images are enlarged without any loss in quality. Additionally, a grid-style submenu for the categories is incorporated into the main section. The header and footer sections remain consistent across the website, with minor adjustments made to the header section to fit the content displayed on each page.

Prototypes:

For the creation of the prototypes, I utilized Figma, as it offers a free tier and provides a comprehensive suite of design tools. I developed functional prototypes for both the mobile and desktop views, accurately replicating the effects and design of the final product. Throughout this process, I adhered strictly to the manager's instructions and the wireframe designs to ensure the prototypes met the specified requirements.

Interface with demo data

Frontend design with hardcoded / demo data:

Following the completion of the wireframes and prototypes, I proceeded to develop the frontend of the application. Below are examples of the code from the Header component of the Next.js project.

At the beginning of the file, I import all the necessary libraries and utilities required for this component. Since the component utilizes React hooks, it must be rendered on the client side. Therefore, the first line of code defines 'use client'. The remaining imports include the mentioned hooks, the React object from the React library, icons from lucide-react, the Image component from Next.js, and my custom SearchComponent.

In the component definition, I initialize two state variables to store temporary information based on user interactions. To optimize performance, I utilize the useEffect hook to ensure that React does not unnecessarily re-render the page when the values of these variables change, which would otherwise cause the variables to be reset. I include the variables in the dependency array of the useEffect hook to track changes made to these specific variables.

In the rendering of the component, I create a wrapping <div> element for the entire component, which utilizes display: flex with a column direction and expands to the full width of the viewport. Next, I define an element to hold the upper portion of the menu. This element also employs display: flex, but with a row direction for its child elements. The content is justified with space-between, ensuring even spacing between elements, and items are centered using align-items: center.

When the screen width reaches the breakpoint for large screens (@media (min-width: 1024px)), the direction changes to column, stacking the items vertically. By default, for screens with a maximum width of less than 1024px, the top-level menu is hidden. At the 'lg' breakpoint, the element reverts to display: flex, with the primary change being its height, which is set to 50px.

The main theme of the component is the use of display: flex, which efficiently handles the alignment of elements in a single dimension, either horizontally or vertically.

The management of breakpoints is handled using the standardized values provided by Tailwind CSS. It is important to note that Tailwind CSS adopts a 'mobile-first' approach, meaning its media queries are structured as @media (min-width: ...px). Consequently, the design process begins with smaller screens and progressively enhances for larger screens.

Another notable aspect is the mobile menu, which contrasts with the main menu.

On smaller screens, the main menu is hidden. In its place, a hamburger icon is displayed. Upon interaction, this icon triggers the mobile menu, which slides into view from the left with a smooth transition. The mobile menu contains the same categories as the main menu. The wrapping element of the mobile menu prevents vertical scrolling of the site to avoid user confusion.

On larger screens, the mobile menu and the hamburger icon are hidden. Instead, the main menu is displayed, featuring smooth effects upon mouse interactions such as hover, click, and area leave.

Dynamic data

To enable my frontend (Next.js) to retrieve data from the backend (Directus), I made several adjustments to accommodate the Directus API. First, I added the necessary information about my frontend to the docker-compose.yml file. Next, I installed the Directus SDK npm package using the command npm install @directus/sdk.

After the installation, I created a new file named directus.ts in the libraries folder of my frontend project. In this file, I imported the required methods and defined an interface for the categories, as these were the initial data integrated into the site. I also created a schema interface to describe the structure of the database, focusing on the Category collection since it was the only table used.

I then instantiated Directus and developed helper functions to fetch the necessary data from the database. Finally, I exported the Directus instance for use throughout the project.

To display the categories in my menu, I imported the getCategories helper function and the Category interface from the directus.ts file into my Header component. I then created a function to retrieve data using the getCategories function and invoked it within a useEffect hook to ensure the request is executed only once. I defined a new variable to store the returned data and created a hash map for my icons, allowing me to display the correct icon for each corresponding category.

Finally, in the JSX section where the menu items are rendered, I used a map function to iterate over the categories and return their contents.

Activity type 2:

Diagram for database

I created the database diagram using Draw.io. The database comprises several tables relevant to e-commerce, including a products table, categories table, tags table, and related connecting tables. It is important to note that many-to-many relationships require connecting tables, while one-to-many relationships only require foreign keys.

For the products, a gallery of product images is necessary. Therefore, I created an additional table connected to the products table. This table contains the image URLs, names, alt text for the corresponding product, and the foreign key linking each image to the correct product.

The products table includes standard fields such as name, ID, description, price, and SKU. Additionally, it contains fields for stock number and stock status for better inventory management, weight and dimensions for shipping management, and meta information to aid in on-site and platform searches, such as Google Search.

The tags and categories tables are utilized for improved product management and to facilitate easier product searches.

Creation of database

For the backend, I chose Directus, a well-known and user-friendly backend system based on Node.js. I followed the self-hosting guide provided in the Directus documentation for the installation.

I began by downloading Docker, as the project needed to be containerized to run. The installation process was straightforward, requiring only a few clicks to complete.

After installing Docker, I created a Directus folder for my project, which included three subfolders: database, uploads, and extensions. In the main project folder, I created a file named docker-compose.yml containing the necessary instructions for Docker. Finally, I ran the Docker container by navigating to the folder containing my files and executing the command docker compose up.

Once the download was complete, I could access my backend via either http://localhost:8055 or http://127.0.0.1:8055 .

Creating tables and relations

After setting up the Directus backend in Docker, I proceeded to create the database tables (referred to as collections in Directus) and define their relationships. To launch Directus, the Docker container must be running. I typically use the Docker interface to start my container, though other methods are available as mentioned previously.

Once the Docker container is running, I access the Directus interface by navigating to the localhost URL and entering my credentials. Upon successful login, I create a new table by navigating to the 'Settings' section at the bottom of the left side menu, selecting 'Data Model,' and clicking the '+' icon to create a new collection.

In the subsequent section, I specify a name for the collection, determine whether it should be treated as a single object, and set the primary key. I then add any additional fields by accessing the 'Optional Fields' tab and clicking the accept icon to confirm my selections.

Next, I select my newly created collection and click the 'Create Field' button to add any necessary fields. This step is crucial as it is where relationships between tables are defined. I choose the field type (for example, a simple input field), fill in the required information, and save the changes.

My new collection is now ready for use. If further modifications are needed, I can repeat the process, edit existing fields, or add complementary information such as automatic sorting based on a specific field.

To add content to my new collection, I navigate to the 'Content' tab in the left side menu, select my collection, and click the 'Create Item' button. I then set the status of the item, fill in the fields, and save the entry by clicking the save button.

This workflow was typical during the development of my internship project.