Webstore Development and Architecture Details

1 Summary

Znode Webstore front-end development primarily involves creating the user interface and ensuring smooth interactions with the back-end through APIs (Application Programming Interfaces). This includes designing visually appealing and intuitive product pages, shopping carts, and checkout processes. Connecting with APIs enables the webstore to retrieve and update data from the server, facilitating dynamic content, real-time updates, and seamless transactions. Attention to user experience, responsiveness across devices, and efficient API integration are key aspects of successful webstore front-end development.

2 Technologies Used in the Webstore

Note: There is no need for a standalone installation for these elements; the CLI will automatically handle the installation of the necessary libraries or required dependencies.

- **Node.js and Next.js:**
  - Utilize Node.js as the runtime environment.
- **Next.js:**
  - Employ Next.js as the framework for web application development.
- **HTML, CSS, Tailwind CSS:**
  - Implement HTML for structuring web content.
  - Use CSS and Tailwind CSS for styling, ensuring a visually appealing and responsive design.
- **Redis:**
  - Integrate Redis as a caching mechanism.
  - Enhance performance through efficient storage and retrieval of frequently accessed data.
- **JavaScript and React:**
  - Leverage JavaScript for scripting and interactivity.
  - Utilize React, a JavaScript library, for building dynamic and interactive user interfaces in the webstore.
- **NX:**
  - Nx for managing the monorepo, Pagebuilder & Webstore.
- **IDE:**
  - Visual Studio Code

3 Development Setup

Follow these steps to set up your local development environment for a Znode Webstore using Node.js, Next.js, HTML, CSS, Tailwind CSS, Redis, and React.

3.1 Prerequisites

- **Node.js Installation:**
  - Download and install Node.js from https://nodejs.org/en
- **Visual Studio Code (VSCode) Installation:**
  - Download and install VSCode from https://code.visualstudio.com.
  - Install the following optional extensions for code quality management(Optional):
    - Code Spell Checker
    - Bundle Size
    - Git History
    - Prettier - Code formatter
    - ES - Lint
    - Sonar Lint

3.2 Znode CLI Tool

- Follow the Znode 10 CLI setup document to download the Webstore SDK on your local machine. Refer Znode 10 CLI Setup document for more details

3.3 Project Setup

- **Navigate to Project Directory:**
  - Open your terminal and move into the project directory using the following command:

```
cd <project_directory>
```

- Replace `<project_directory>` with the name of the directory created during the Znode 10 CLI installation process.

- **Install Dependencies**
  - Run the following command to install project dependencies:

```
npm install
```

  - Alternatively, use `yarn dev` if you're using Yarn.

3.4 Configure Environment Variables:

- In the root directory of the project, update the .env file with the following keys and corresponding values:

```
# General Settings
NEXTAUTH_URL="http://localhost:3000"
NEXTAUTH_SECRET="ABCDEF"
WEBSTORE_DOMAIN_NAME="http://localhost:3000/"
APP_NAME="WEBSTORE"
IS_DEBUGGING=false
DEFAULT_THEME="base"
CONTENT_SECURITY_POLICY=""

# API Configuration
API_URL="https://apigateways-z10-qa.amla.io/"
API_DOMAIN="api-z10-qa.amla.io"
API_V2_DOMAIN="api-v2-qa-znode.amla.io"
API_KEY="a2c32d26-fb56-4e2d-aa44-10ae401a0970"

# Payment Manager
NEXT_PUBLIC_PAYMENT_MANAGER_URL="https://znode10stage.azureedge.net/plugin-payment/PaymentManager.js"

# Google API Configuration
NEXT_PUBLIC_GOOGLE_MAP_API_KEY="AIzaSyDzQcVG1LYGzI8U7IlQwuZcl8S6eSxRLzc"
```

```
# Security Configuration
NODE_TLS_REJECT_UNAUTHORIZED=0

# Logging and Caching
LOGGING_METHOD="FILE"   # or "API"
CACHE_MEMORY="FILE" # or "REDIS"
ENABLE_CACHE=true
ENABLE_PAGE_CACHE=false

# Redis Configuration
REDIS_CONNECTION_STRING=""

NEW_RELIC_APP_NAME="Z10-WEBSTORE"
NEW_RELIC_LICENSE_KEY="249f803b034b74a3ffd69cacb80ab69cFFFFNRAL"

# Storage Configuration
STORAGE="FILE" # "REDIS" | "FILE" | "CDN" is used for Cache Storage
```

```
STORAGE = "file"
REDIS_CONNECTION_STRING = ""
APP_NAME = "WEBSTORE"
IS_DEBUGGING = false
```

- **NEXTAUTH_URL**
  Base URL for NextAuth authentication callbacks.
  Example: http://localhost:3000

- **NEXT_PUBLIC_PAYMENT_MANAGER_URL**
  URL for the Payment Manager service (exposed to the client).
  Example: https://payment-service.com

- **NODE_TLS_REJECT_UNAUTHORIZED**
  Disables SSL validation (use only for local development).
  Example: `0`

- **API_URL**
  Base URL for the backend API.

Example: https://backend-api.com/api

- **NEXTAUTH_SECRET**
  Secret key for encrypting JWTs and securing cookies.
  Example: `ABCDEF`

- **WEBSTORE_DOMAIN_NAME**
  Domain name for the webstore.
  Example: `localhost:3000`

- **API_DOMAIN**
  Domain for accessing the primary API.
  Example: `api-domain-name`

- **API_V2_DOMAIN**
  Domain for accessing version 2 of the API.
  Example: `api-v2-domain-name`

- **API_KEY**
  API key for authenticating requests.
  Example: `******`

- **LOGGING_METHOD**
  Method for logging application events (`FILE` or `CONSOLE`).
  Example: `FILE`

- **CACHE_MEMORY**
  Cache storage type (`MEMORY` or `FILE`).
  Example: `FILE`

- **ENABLE_CACHE**
  Flag to enable or disable caching.
  Example: `true`

- **NEXT_PUBLIC_GEO_CODE_URL**
  URL for geocoding services (exposed to the client).
  Example: https://geo-service.com

- **NEXT_PUBLIC_GOOGLE_MAP_API_KEY**
  API key for Google Maps services (exposed to the client).

Example: `******`

- **CONTENT_SECURITY_POLICY**
  Defines allowed content sources for security.
  Example: `default-src 'self'`

- **X_CONTENT_TYPE_OPTIONS**
  Prevents MIME-type sniffing by the browser.
  Example: `nosniff`

- **PERMISSIONS_POLICY**
  Specifies permissions for APIs like camera, fullscreen, and geolocation.
  Example: `camera=(self), fullscreen=(self), geolocation=(self)`

- **REFERRER_POLICY**
  Controls referrer information included with requests.
  Example: `strict-origin-when-cross-origin`

- **STRICT_TRANSPORT_SECURITY**
  Enforces secure (HTTPS) connections.
  Example: `max-age=31536000; includeSubDomains; preload`

- **DEFAULT_THEME**
  Sets the default theme for the webstore.
  Example: `base`

- **STORAGE**
  Specifies storage type used by the application.
  Example: `file`

- **REDIS_CONNECTION_STRING**
  Connection string for Redis database.
  Example: `redis://localhost:6379`

- **APP_NAME**
  Name of the application.
  Example: `WEBSTORE`

- **IS_DEBUGGING**
  Enables or disables debugging mode.
  Example: `false`

3.5 Access the local dev environment:

- Ensure you are in the root directory of the project. This is where the main configuration files (e.g., `package.json`, `nx.json`, or `next.config.js`) are located. You can navigate to the root

```
npx nx dev <project-name>
```

- **Example:** To run the `webstore` project:

```
npx nx dev webstore
```

- Open your web browser and go to the specified localhost address (usually http://localhost:3000).

  Note: Ensure that there is an entry in the Znode domain database table for seamless integration

3.6 Access the local Page builder environment:

- Ensure you are in the root directory of the project. This is where the main configuration files (e.g., `package.json`, `nx.json`, or `next.config.js`) are located. You can navigate to the root

```
npx nx dev <project-name>
```

- **Example:** To run the `pagebuilder` project:

```
npx nx dev page-builder
```

- Open your web browser and go to the specified localhost address (usually `http://localhost:3000/en-US/page-builder?url=header&theme=base&publishState=preview&themeCode=base&storeCode=MaxwellsHardware&pageCode=Header`).

```
http://localhost:3000/en-US/page-builder?url=header&theme=base&publishState=preview&themeCode=base&storeCode=MaxwellsHardware&pageCode=Header
```

Query Parameters

Parameter	Description
url	Specifies the component/page being edited (header, footer, etc.).
theme	Defines the theme (base, dark-mode, etc.).
publishState	Controls the visibility (preview or published)
themeCode	Internal reference for the theme.
storeCode	Identifies the store (MaxwellsHardware, etc.)
pageCode	Specifies the page section (Header, Footer, etc.).

Note: Ensure that there is an entry in the Znode domain database table for seamless integration

3.7 Access the production build on local environment:

- Ensure you are in the root directory of the project. This is where the main configuration files (e.g., `package.json`, `nx.json`, or `next.config.js`) are located. You can navigate to the root

```
npx nx start <project-name>
```

- **Example:** To run the `webstore` project:

```
npx nx start webstore
```

- Open your web browser and go to the specified localhost address (usually http://localhost:3000).

4 Znode Webstore Application

4.1 Folder Structure

The project is organized into several key directories:

.github

Contains GitHub-related configurations, typically for **GitHub Actions**, CI/CD workflows, and issue/PR templates.

- **workflows**: Define automated processes like testing, linting, and deploying the project when code is pushed or merged into certain branches.

.husky

This directory manages Git hooks, particularly **pre-commit** and **pre-push** hooks. These hooks enforce standards such as running tests, linters, or formatting before the code is committed or pushed to the repository.

- **pre-commit**: Likely runs linters and formatters to ensure code quality.
- **pre-push**: Runs tests to ensure that the pushed code does not break the application.

.nx

Configuration for the Nx workspace. Nx helps in managing the monorepo by allowing developers to build and test specific parts of the application independently. Nx is key for orchestrating tasks like building, testing, and serving the apps.

.vscode

This folder holds workspace-specific settings for **Visual Studio Code**. It can include settings for:

- Formatting and linting
- Debugging configurations
- Extensions recommended for developers working on this project.

apps

Contains the core applications of the project. Each application is a standalone project and can be built, tested, and served independently.

- **page-builder**:
    - This application is responsible for creating and managing dynamic pages in the webstore.
    - Developers working on this application will likely be dealing with content management features, drag-and-drop interfaces, and API integrations for storing and retrieving content.
- **webstore**:
    - The customer-facing ecommerce application.
    - This app will contain most of the storefront features, product displays, shopping cart functionality, and checkout processes.

node_modules

This directory contains all the installed dependencies for the project. It is generated automatically by package managers (npm/yarn) and should not be modified manually. It's recommended to add this folder to .gitignore to prevent versioning unnecessary fil**packages**

Contains shared libraries, utilities, and components used by multiple apps in the monorepo. This helps in maintaining DRY (Don't Repeat Yourself) principles by promoting code reuse across different applications.

- **agents**:
    - Handles API calls, likely using fetch or Axios. This package could contain services for handling communication between the client and backend.
- **base-components**:
    - This package includes reusable UI components such as buttons, inputs, modals, etc., which are shared across both the page-builder and webstore.
- **clients**:
    - This likely manages third-party integrations, such as payment gateways, authentication providers, or external APIs and client side api that will call the agent.
- **constants**:
    - A package that holds common constants, such as environment variables, default values, configuration settings, or API endpoint URLs.
- **custom-theme1-components / custom-theme2-components / theme1-components**:
    - These packages represent custom themes for the ecommerce store. They provide styled components and layout variations that can be applied to different versions of the webstore. Theme customization allows for visual flexibility.
- **logger**:
    - Centralized logging package that defines the format and structure of logs (e.g., error logging, info logs). This helps in debugging and auditing.
- **page-builder**:
    - Additional page builder utilities or components. This could provide higher-level logic specific to the page-builder application.
- **types**:
    - Shared TypeScript types and interfaces used across the project. Having a centralized types package ensures that all applications share the same type definitions and interfaces.
- **utils**:
    - Common utility functions like data manipulation, validation, or formatting. These are helpers that can be used across different applications and packages.

4.2 Key Configuration Files

.eslintrc.json / .eslintignore

Configuration for **ESLint**, which helps in identifying and fixing common coding problems and enforcing coding standards. The .eslintignore file is used to specify which files or directories ESLint should ignore during linting.

.prettierrc / .prettierignore

Configuration for **Prettier**, which is used for automatic code formatting. The .prettierignore file lists files or directories that should be excluded from Prettier formatting.

tsconfig.base.json

This is the root TypeScript configuration file for the project. It includes paths and base configurations used by different apps and packages to ensure they share the same TypeScript settings.

nx.json

Configuration file for **Nx**, which defines how the workspace is managed. It includes settings for tasks like caching, building apps, testing, and serving.

package.json

Contains metadata about the project as well as the scripts that can be run for development (e.g., start, build, test). It also includes dependencies and devDependencies required by the project.

Refer folder structure diagram here

5 Architecture Details

Refer to the Znode 10 Webstore Architecture document.

6 Theme Setup

6.1 How to Create the Theme

To create your theme component, you need to run the following command:

```
npx nx g @nx/next:lib packages/{themeName}
```

This command will create your theme component package. After that, you can create your custom components inside this package.

File Structure

The recommended file structure should follow this convention:

- **Create a `widgets` folder**: This will serve as the main directory for organizing your components. Inside this folder, create subfolders based on widget usage, such as:
    - `UI-widgets`
    - `Page-widgets`
    - `Znode-widgets`

6.2 How to Create and Configure a Component in Page Builder

This section outlines the detailed steps for creating new components and integrating them into the Page Builder system. Components can be categorized into the following areas based on their purpose and usage

When developing new components, you must determine where they belong within the project structure:

- **Znode-widgets**

    - Znode widgets are components that rely on Znode configuration, such as the Banner Slider, Content Blocks, etc. These components are decoupled and primarily depend on configuration elsewhere in the CMS or Admin Console. This classification of widgets allows developers to easily identify the impact based on the configuration.
- **Ui-widgets**
    - UI widgets are custom widgets primarily created for the user interface, focusing on the look and feel of the theme. For example, consider a Heading component that is commonly used across the site. Developers can create such granular components for reusability. Another example is an Image widget. These types of widgets can be created to allow end users or admin users to easily use them to build custom pages.
- **Page-widgets**
    - Page widgets are widgets that define the overall structure or wrapper of a page. These widgets mainly handle the layout of the page and include essential elements needed for the page's functionality.

The following guide provides an example for creating a new **UI widget** component, such as a grid component.

Steps for Creating a New Component

2. **Identify the Component Area**
   Decide whether the component belongs to `znode-widgets`, `ui-widgets`, or `page-widgets` based on its purpose and usage.
3. **Create the Component Folder**
   For a new **UI widget** component, navigate to:

```
packages/page-builder/src/configs/base-config/widgets/ui-widgets/
```

   Create a folder named after your component. For this example, create a folder called `grid`.

6. **Add Component Files**
   Inside the newly created `grid` folder, create the following files:
    a. **GridConfig.tsx**
    b. **GridRender.tsx**

7. **Configure `GridConfig.tsx`**
   The `GridConfig.tsx` file is responsible for defining the component's configuration settings
   For example:
   - Number of grids
   - Row size
   - Column size

8. **Develop `GridRender.tsx`**
   The `GridRender.tsx` file handles rendering the component's UI based on the configurations provided in `GridConfig.tsx`.

9. **Register the Component**
   Once the component is created, configure it in the `base-components-config.tsx` file. This ensures the new component is available within the Page Builder system.

   Note:
   - ➔ The above steps apply only to base-config components.
   - ➔ If you are working on theme-level components, follow the theme-specific creation and configuration guidelines instead**.**

6.3 How to update the component and configure in page builder

This section provides guidelines for updating existing components in the Page Builder system, whether for adding new settings or modifying the user interface.

Steps for Updating an Existing Component

1. Identify the Component
   Determine the component you want to update (e.g., Grid component).

2. Decide the Type of Update
   Based on your requirements, identify the area of change:

   a. Add/Modify/Remove Settings
   b. Update the UI

Update/Remove Settings in a Component

1. File to Update
   Locate the `Config` file of the component. For example, for the `Grid` component:

   ```
   packages/page-builder/src/configs/base-config/widgets/ui-widgets/grid/GridConfig.tsx
   ```

2. Steps
   a. To Add a New Setting
      Define the new setting in `GridConfig.tsx`.
      Example: Adding a setting for "grid spacing" or "background color".
   b. To Remove a Setting
      Simply delete the corresponding configuration field from `GridConfig.tsx`.

Update the Component UI

1. File to Update
   If the changes involve UI updates, navigate to the `Render` file. For the `Grid` component:

   packages/page-builder/src/configs/base-config/widgets/ui-widgets/grid/GridRender.tsx

2. Steps:
   a. Modify or enhance the UI as needed in `GridRender.tsx`.
   b. Implement new design elements, adjust layout styles, or add interactive features.

Note: Since the component is already configured in the `base-components-config.tsx` file, no additional configuration is needed after updating the component.

6.4 How to create a theme component in page builder

This section provides a step-by-step guide for creating and configuring theme components within the Page Builder package.

Steps for Creating a Theme Component

1. **Create a Theme Folder**
   - Navigate to the `configs` directory inside the `page-builder` package.

- Create a new folder for your theme. For example:

```
packages/page-builder/src/configs/bstore-config
```

2. **Create `root-config.ts` File**
   - Inside the theme folder, create a `root-config.ts` file.
   - This file will inherit base configuration components such as `znode-widgets`, `ui-widgets`, and `page-widgets`.

3. **Extend Base Configurations**
   - Use the `extendConfig` method (provided by the base configuration) to inherit the required widgets.
   - Create a `getRootConfig` method in `root-config.ts` to call the `extendConfig` method.

```
import { extendConfig } from '../base-config';

export const getRootConfig = () => {
  return extendConfig({
    // Inherit widgets from base config
  });
};
```

4. **Override Existing Widgets**
   - If you need to customize any widgets from the base configuration, you can override them in the `root-config.ts` file.

5. **Add New Widgets**
   - To introduce new widgets, create them in the appropriate folder (e.g., `ui-widgets` or `page-widgets`).

    - Update the `root-config.ts` file to include the new widgets.

Map the Theme in `get-config.ts`

- After creating and configuring the theme, map it in the `get-config.ts` file located in the `root-configs` folder.
- This file acts as the root configuration loader for the project.

```
import { getRootConfig as getBstoreConfig } from
'./bstore-config/root-config';

export const getConfig = () => {
  return getBstoreConfig();
};
```

```ts
export function getRootConfig(params: IConfigParam) {
  const override = {
    // ** Override your components or add
    Button: ButtonConfig,
    Card: CardConfig,
  };

  // Mapping of config types to their respective page configs
  const configMap: { [key: string]: any } = {
    category: { ProductListPage: ProductListPageConfig },
    product: { ProductDetailsPage: ProductDetailsPageConfig },
    // Add more config types as needed
  };

  // Fetch the appropriate overrideConfig based on the configType
  const overrideConfig = configMap[params.configType] ? { ...configMap[params.configType], ...override } : override;

  return extendConfig({
    configParams: params,
    overrideConfig: overrideConfig,
  });
}

```

```ts
import { getRootConfig as getBstoreRootConfig } from "./bstore-config/config/root-config";

const themeConfigMap = new Map([
  ["common", getBaseRootConfig], // common and core config, which is use to every custom theme or override.
  ["theme1", getTheme1RootConfig],
  ["theme2", getTheme2RootConfig],
  ["safetygear", getSafetyGearRootConfig],
  ["bstore", getBstoreRootConfig],
  // ** add your theme config
]);

export function getConfig(params: IConfigParam) {
  const rootConfig = themeConfigMap.get(params.theme.toLowerCase());

  if (!rootConfig) {
    return getBaseRootConfig(params);
  }

  return rootConfig(params);
}

```

6.5 How to update theme component and configure in page builder

To update the theme component or override the existing base component, you should go to your theme config. For example:

1. Decide which component you want to update. And decide which area you are looking for change for example whether you want to add a new setting in existing components or UI changes.
2. If you want to add a new setting in an existing component, for example, add a new setting in Grid component, so you have to go GridConfig.tsx, in this file you can add a new setting.

NX command details

1. Run Tasks: These commands run tasks on your code.
    a. Run Webstore: `npx nx run webstore`
    b. Build Webstore: `npx nx run webstore:build`
    c. Lint Webstore: `npx nx run webstore:lint`
    d. Run Page-Builder: `npx nx run page-builder`
    e. Build Page-Builder: `npx nx run page-builder:build`
    f. Lint Page-Builder: `npx nx run page-builder:lint`
    g. Clear Nx Cache: `npx nx reset`

6.6 Why two apps in project

Webstore

The Webstore is the frontend application that renders the pages created and published via the Page Builder. It dynamically fetches the published data and displays it to the end-users, allowing them to browse products, view promotional content, and interact with various widgets on the site.

Features of the Webstore

1. **Dynamic Content Rendering**: Webstore fetches and renders content dynamically from the backend, including product listings, banners, images, and promotional content created in the Page Builder.
2. **Responsive Design**: The Webstore is designed to be fully responsive, ensuring that it works well on a variety of devices including desktops, tablets, and mobile phones.

3. **Real-Time Data**: Whenever a new page is published or updated in the Page Builder, the Webstore automatically pulls the latest content to ensure users always see up-to-date information.
4. **Integrated Widgets**: Widgets created in the Page Builder (e.g., product carousels, promotional banners) are rendered and interacted with directly in the Webstore.
5. **Internationalization (i18n)**: Supports multiple languages and regions, allowing you to present content tailored to different markets.

Page builder

The **Page Builder** is an admin application designed to allow users to create and manage pages for the webstore via a drag-and-drop interface. It allows users to design custom pages by selecting and arranging components and widgets, then publishing the page content to the backend. Once a page is published, the data is stored in the backend and can be rendered in the **Webstore** application.

Features

1. **Drag and Drop Interface**: Users can easily create and design pages by dragging and dropping pre-defined components onto the page layout.
2. **Widgets**: Various widgets such as UI widgets (e.g., buttons, cards, images) and Znode-specific widgets (e.g., product carousels, banners) are available to customize the page content.
3. **Page Publishing**: After designing the page, the user can publish the content, which will be saved in the backend and made available for the Webstore.
4. **Real-time Preview**: Users can preview the pages before publishing to see how the design looks with the content.

6.7 Create a new page in page builder

The Page Builder provides flexibility for creating new pages through two main approaches: manual creation or selecting from predefined templates. Here's a guide to creating a new page using both methods.

1. Page Creation Using Dropdown Selection
    a. This method simplifies page creation by allowing users to select from a list of predefined page templates.

b. **Select a Template**: Choose a predefined template from the dropdown list that suits the page you want to create (e.g., Home Page, About Page, Product Page).
c. **Configure Page Settings**: After selecting the template, configure additional settings such as:
    i. Page title
    ii. Route or URL (e.g., `/about-us`, `/products`)
d. **Customize Components**:
    i. Use the drag-and-drop functionality of the **PageEditor** to customize the layout and components of the page.
    ii. Adjust component properties as needed.
e. **Preview and Save**:
    i. Preview the page to ensure it meets requirements.
    ii. Save the page to publish it in the database or make it available for the frontend.

6.8 Create a new page without page builder

If you choose to create a page outside the Page Builder, you can use the traditional Next.js method to manually define and build the page. This approach is ideal for pages requiring highly customized functionality or structure that does not depend on the Page Builder's drag-and-drop or dynamic capabilities.

Steps to Create a New Page Without Page Builder

1. **Navigate to the Pages Directory**: Open your Next.js project and locate the `pages` directory where all page files are stored.
2. **Create a New Page File**: Add a new file in the `pages` directory or a subfolder, following Next.js routing conventions.
3. **Define the Page Component:** Write the React component for your page. This will include all the necessary logic, UI, and functionality.
4. Add Server-Side Logic (Optional)

Tailwind Updates

To create a custom theme in Tailwind CSS, you can follow a structured approach. A theme defines unique design tokens and configurations for your specific branding or styling needs while maintaining compatibility with other themes in the project. Below is a detailed explanation:

Steps to Create a Theme

1. **Create a tailwind.config.ts**
    a. Each theme should have its own tailwind.config.ts file located in its respective theme package.
    b. For example, for a bstore theme; bstore->tailwind-config/tailwind-config.js

2. **Define the Theme Object**
    a. The theme configuration should be an object that includes the following properties:
        i. Name key: A **name** key whose value is a unique identifier for your theme. This name is used internally to differentiate themes. Example: "bstore"
        ii. Selectors key: An array of selectors used to apply the theme. These selectors determine which elements or parts of the application the theme applies to. Example: ["[data-theme='bstore']"]
        iii. Extend key: This property allows you to add new design tokens or override existing ones. Important: Avoid overriding default Tailwind tokens directly, as it may unintentionally affect other themes.

3. **Configure theme with webstore/PageBuilder**
    a. Configure your Tailwind CSS setup to support multiple themes by adding the themes plugin. This configuration allows you to manage and switch between multiple theme-specific Tailwind settings, such as bstoreTailwindConfig and safetyGearTailwindConfig.
    b. Example, in webstore/tailwind.config.js or page-builder/tailwind.config.js

```
const { createGlobPatternsForDependencies } = require("@nx/react/tailwind");
const { join } = require("path");
const themes = require("tailwindcss-themer");

const tailwindConfig = require(join(__dirname, "../..", "packages/base-components/src/tailwind-config/tailwind.config.js"));
const safetygearTailwindConfig = require(join(__dirname, "../..", "packages/safetygear/tailwind-config/tailwind.config.ts"));
const bstoreTailwindConfig = require(join(__dirname, "../..", "packages/bstore/tailwind-config/tailwind.config.ts"));
// import your theme package

/** @type {import('tailwindcss').Config} */
module.exports = {
  presets: [tailwindConfig],
  content: [join(__dirname, "{src,pages,components,app}/**/*!(*.stories|*.spec).{ts,tsx,html}"), ...createGlobPatternsForDependencies(__dirname)],
  theme: {
    extend: {},
  },
  plugins: [
    themes({
      themes: [
        bstoreTailwindConfig,
        safetygearTailwindConfig,
        // add your custom theme
      ],
    }),
  ],
};
```

```
const themeObj = {
  name:"bstore",
  selectors: ["[data-theme='bstore']"],
  extend: {} // tailwind extend configuration
};
```

6.9 Page builder config details

This documentation provides an overview of how the `baseComponentsConfig` function is set up in a Page Builder configuration. This configuration integrates both UI and Znode widgets and

provides a structure for rendering components dynamically in the page builder. Please refer below screenshots:

Znode Widgets: These widgets are custom components specifically for Znode. They include various components for displaying product carousels, banners, and more.

UI Widgets: These are general-purpose UI components that can be used in the layout of the page. They provide various UI elements such as buttons, text, images, and other layout components.

The `baseComponentsConfig` Function

This function is responsible for generating the configuration of the page builder. It takes an object of type `IBaseConfigParams` containing `header` and `footer` as parameters and returns the configuration for the root and components.

The `root.render` method wraps the `children` of the page in the `header` and `footer` that are passed in as parameters. This allows you to dynamically inject a page's header and footer into the layout.

Components Configuration

The components configuration section defines which widgets are available for use within the page builder. It is organized into three main categories:

1. **UI Widgets**: These are general-purpose UI components that can be used in any part of the page layout.
    a. `VerticalSpacing`: Vertical space for layout.
    b. `Column`: A container for content in a column layout.
    c. `Flex`: A flexible container for components.
    d. `Text`: Basic text component.
    e. `Card`: A component for displaying content in a card format.
    f. `Hero`: A hero section component.
    g. `Logo`: Displays a logo.
2. **Znode Widgets**: These are specialized widgets related to Znode, such as product carousels, banners, and image components.
    a. `BannerSlider`: A slider for banners.
    b. `CategoriesCarousel`: A carousel for displaying categories.
    c. `OfferBanner`: A banner displaying offers.

    d. `ProductsCarousel`: A carousel for displaying products.
    e. `AdSpace`: A space for displaying ads.
    f. `HomePagePromo`: A promotional component for the home page.
    g. `Image`: An image component.
    h. `Video`: A video component.

Categories Configuration

The `categories` section organizes the widgets into logical groups:

1. **Layout Widgets**: Widgets that are used to structure the layout of the page (e.g., `Flex`, `Column`, `VerticalSpacing`).
    a. `components`: An array of component names available in this category.
    b. `title`: The title for this category, shown in the page builder interface.
2. **UI Widgets**: Components focused on UI elements such as text, cards, logos, and buttons
    a. `components`: An array of component names for UI elements.
    b. `title`: The title for this category.
3. **Znode Widgets**: Components specific to Znode functionality, like carousels, product displays, and banners.
    a. `components`: An array of component names related to Znode functionality.
    b. `title`: The title for this category.

This configuration allows the page builder to render and manage a wide range of widgets and components. By defining the components and categorizing them, the page builder interface can dynamically generate the available components based on the configuration. This approach makes it easy to add new components, override existing ones, or extend the functionality as needed.

1. **Znode Widgets**: Specialized components related to Znode.
2. **UI Widgets**: Generic UI components for building pages.
3. **Root Configuration**: Renders the page with a header, content, and footer structure.

This configuration serves as the foundation for extending and customizing the page builder according to the needs of the application.

```tsx
// ***Znode Widgets
import {
  AdSpaceConfig,
  BannerSliderConfig,
  CategoriesCarouselConfig,
  HomePagePromoConfig,
  ImageConfig,
  OfferBannerConfig,
  ProductsCarouselConfig,
  VideoConfig,
} from "../widgets/znode-widgets";
// ***UI Widgets
import {
  ButtonGroupConfig,
  CardConfig,
  ColumnConfig,
  FlexConfig,
  HeadingConfig,
  HeroConfig,
  LogoConfig,
  TextConfig,
  TextImageConfig,
  VerticalSpaceConfig,
} from "../widgets/ui-widgets";
import type { IComponentCategories, IComponentPropsWithoutPages, IRootProps } from "../../../types/page-builder";

import type { Config } from "@measured/puck";
import type { ReactNode } from "react";

interface IBaseConfigParams {
  header: ReactNode;
  footer: ReactNode;
}

export type IBaseComponentsConfig = Config<IComponentPropsWithoutPages, IRootProps, IComponentCategories>;

export function baseComponentsConfig(params: IBaseConfigParams): IBaseComponentsConfig {
  return {
    root: {
      render: ({ children }: { children: ReactNode }) => {
        return (
          <>
            {params.header}
            {children}
            {params.footer}
          </>
        );
      },
    },
    components: {
      // UI Widgets
      VerticalSpacing: VerticalSpaceConfig,
      Column: ColumnConfig,
      Flex: FlexConfig,
      Text: TextConfig,
      Card: CardConfig,
      Hero: HeroConfig,
      Logo: LogoConfig,
      Heading: HeadingConfig,
      ButtonGroup: ButtonGroupConfig,
      TextImage: TextImageConfig,

      // Znode Widgets
      BannerSlider: BannerSliderConfig,
      CategoriesCarousel: CategoriesCarouselConfig,
      OfferBanner: OfferBannerConfig,
      ProductsCarousel: ProductsCarouselConfig,
      AdSpace: AdSpaceConfig,
      HomePagePromo: HomePagePromoConfig,
      Image: ImageConfig,
      Video: VideoConfig,
      // TextEditor: TextEditorConfig,
    },
    categories: {
      layoutWidgets: {
        components: ["Flex", "Column", "VerticalSpacing"],
        title: "Layout",
      },
      uiWidgets: {
        components: ["Text", "Card", "Logo", "Heading", "ButtonGroup", "Hero", "TextImage"],
        title: "UI Widgets",
      },
      znodeWidgets: {
        components: ["BannerSlider", "CategoriesCarousel", "OfferBanner", "ProductsCarousel", "AdSpace", "HomePagePromo", "Image", "Video"],
        title: "Znode Widgets",
      },
    },
  };
}

```

Library configuration in tsconfig

The `tsconfig.json` file is a key configuration file used by TypeScript to control the behavior of the TypeScript compiler for your project. It allows you to specify various compiler options, file inclusions, exclusions, and module resolution strategies that guide how TypeScript compiles and resolves your code.

Here is an overview of the main concepts and options within the `tsconfig.json`:

1. **Compiler Options**
    a. **`target`**: Specifies the JavaScript version the TypeScript code should be compiled to. This allows you to choose whether the output should be compatible with older JavaScript versions (like `es5`) or more modern ones (`es6`, `esnext`).
    b. **`module`**: Determines the module system used for imports and exports. You can choose between options like `commonjs` (Node.js default), `es6` (ES module system), or `umd`, which is commonly used for Universal Module Definition.
    c. **`strict`**: Enables a series of type-checking options that can help prevent common programming mistakes. This includes checks for nulls, undefined, and stricter type checking overall.
    d. **`esModuleInterop`**: When enabled, this option helps TypeScript to interoperate with CommonJS modules by allowing default imports from modules that do not have a default export. This is especially useful when working with JavaScript libraries in TypeScript.
    e. **`skipLibCheck`**: Skips type checking of declaration files (e.g., `*.d.ts`). This can improve performance during compilation, especially for large projects with numerous dependencies.
    f. **`outDir`**: Specifies the directory where compiled JavaScript files will be output. This helps organize your build artifacts in a separate directory, like `dist` or `build`.
    g. **`baseUrl`**: Defines the base directory for relative module imports. Setting this helps resolve module paths more effectively, avoiding the need for long relative paths.
    h. **`paths`**: Allows you to define custom module resolution paths. This can simplify the import process in larger projects, letting you map module names to specific file paths.

6.10 Debugging Instructions

Open the vscode terminal, click on the right side top down arrow as shown in screenshot. Now follow step:

1. Open terminal (ctrl+J), now see right top side, you will see dropdown arrow, so click on it.
2. It will open a new popup, list of terminal options such as Powershell, Git Bash, JavaScript Debug Terminal.

3. Now click on JavaScript Debug Terminal, and run your command. `npx nx run webstore`
4. Now it will run localhost, and it will enable the debug mode of your application, if you add the breakpoint of debugger anywhere. It will break on it and you can debug your execution.

How Page Builder works

The **Page Builder** operates as a dynamic platform that enables the creation of pages by leveraging component configurations and pre-stored data. Here's an overview of its workflow:
How Page Builder Works

1. **Component Configuration**:
    a. Component configurations are essential to define the structure and behavior of individual widgets or UI components.
    b. These configurations are passed to the `PageEditor` component, which acts as the primary engine for rendering and managing components within the Page Builder.
2. **Data Integration**:
    a. Data required for the components, such as content, images, or settings, is fetched from a database, CDN, or other storage locations.
    b. This data is seamlessly integrated into the Page Builder to populate components with dynamic content.
3. **PageEditor Component**:
    The `PageEditor` is the core component responsible for rendering the UI and providing drag-and-drop functionality.
    It uses the component configurations and associated data to create an interactive interface where users can:
        i. Drag and drop widgets to build layouts.
        ii. Configure components dynamically.
        iii. Preview the page before publishing.
4. **Page Creation**:
    a. Users interact with the `PageEditor` to design pages by combining various components.
    b. Once the page design is finalized, it can be published, storing the configuration and layout data in the database or CDN for use in production.

6.11 How to Create Component Configurations

Configuration components in the Page Builder serve as the blueprint for individual widgets or UI elements. These configurations define the properties, behavior, and structure of components, enabling them to be integrated into the PageEditor and rendered dynamically.

How Component Registration Works

Steps to Create a Configuration Component

1. **Define the Component**
    a. Start by identifying the widget or component you want to configure (e.g., a button, carousel, or image).
    b. Create a file for the configuration in the appropriate folder, such as `ui-widgets`, `znode-widgets`, or `page-widgets`.
2. **Organize Configuration Files**
    a. Group similar components into folders for better maintainability
        i. **UI Widgets**: Basic visual components (e.g., buttons, text, cards).
        ii. **Znode Widgets**: Specialized widgets for Znode functionalities (e.g., product carousels, banners).
        iii. **Page Widgets**: Components tied to specific page functionalities.
3. Register the Component
    a. Add the new component configuration to the central configuration file (e.g., `baseComponentsConfig`).
4. Define Categories (Optional)
    a. Assign the component to a category for organization within the PageEditor.
    b. Categories determine how components are displayed in the editor's UI.

7 Q&A

1. Why Node.js as the Runtime Environment?registering

Pros:

Efficiency and Scalability: Node.js employs a non-blocking, event-driven architecture, making it highly efficient and scalable for handling multiple concurrent connections.

JavaScript Unification: Node.js enables a unified language (JavaScript) for both frontend and backend development, simplifying codebase management.

Cons:

Single-Threaded Limitation: Node.js is single-threaded, which might pose limitations for CPU-bound tasks.

Alternative: *Java (Spring Boot) & .Net*

Pros: Strong ecosystem, extensive libraries, multi-threading support.

Cons: Heavier resource consumption, and slower startup times compared to Node.js.

2. Why Next.js as the Web Application Development Framework?

Pros:

SSR and SSG: Next.js supports Server-Side Rendering (SSR) and Static Site Generation (SSG), leading to faster page loads and improved SEO.

Structured Project Organization: Next.js provides a well-organized framework, simplifying development and aiding scalability.

Cons:

Learning Curve: Next.js may have a steeper learning curve compared to simpler alternatives, especially for beginners.

Alternative: *React with Create React App (CRA)*

Pros: Simplicity, easy setup, great for smaller projects.

Cons: Lacks built-in SSR and SSG, which Next.js provides for improved performance.

3. Why HTML for Structuring Web Content?

Pros:

Standardization: HTML is the standard markup language for structuring content, ensuring compatibility across different browsers and devices.

Semantic Structure: HTML's semantic elements contribute to a well-organized and accessible webstore.

Cons:

Limited Interactivity: HTML alone may have limitations in providing complex interactivity compared to more modern Single Page Applications (SPAs).

Alternative: *Markdown (MDX)*

Pros: Simplicity, easy to learn, human-readable.

Cons: Limited in complex layouts compared to HTML.

4. Why CSS and Tailwind CSS for Styling?

Pros:

Responsive Design: CSS is crucial for styling, and Tailwind CSS provides a utility-first approach for efficient and responsive designs.

Consistency and Efficiency: Tailwind CSS's utility classes enable a consistent and efficient styling process.

Cons:

Learning Curve: Tailwind CSS may have a learning curve, and some developers may find it more verbose compared to alternatives like SASS.

Alternative: *SASS/SCSS with Bootstrap*

Pros: Bootstrap provides a comprehensive UI toolkit.

Cons: Heavier file size, less customization compared to Tailwind CSS.

5. Why Redis as a Caching Mechanism?

Pros:

Performance Optimization: Redis is a high-performance, in-memory data store, optimizing webstore performance by caching frequently accessed data.

Fast Data Retrieval: Redis's quick data retrieval reduces response times for users.

Cons:

In-Memory Limitations: Redis operates in-memory, which can pose limitations in handling large datasets.

Persistence Challenges: Achieving persistence in Redis might require additional configurations.

Alternative: *Memcached*

Pros: Simplicity, good for simple key-value caching.

Cons: Limited functionality compared to Redis, less suitable for complex data structures.

6. Why JavaScript and React for Interactivity?

Pros:

Client-Side Interactivity: JavaScript enhances client-side interactivity, contributing to a dynamic and engaging user interface.

Component-Based UI Development: React facilitates the creation of reusable UI components, streamlining development.

Cons:

Heavier Initial Load: React may result in a heavier initial load compared to server-rendered pages.

Steeper Learning Curve: React can have a steeper learning curve, especially for newcomers to frontend development.

Alternative: *Vue.js*

Pros: Simplicity, easy integration, smaller learning curve.

Cons: Smaller ecosystem compared to React