Wix Headless: Building an eCom site workshop

Everything you need to get started with Wix Headless

Our Project Template

🥅 Our Goal: Take an existing Commerce website that is mostly implemented, and add the Wix specific logic to create an amazing, fast and state-of-the-art online Wix store!

We will be using the NextJS Commerce template as our project starter. This website template implements many important features for a commerce site:

- Home page with featured products
- Search products by name
- A product page with product details
- Recommended products next to each product
- Cart functionality with adding, removing and updating a Cart
- A checkout process managed by the commerce provider
- Extra information pages (about, policies etc), managed by a CMS (dynamic menu, dynamic pages)

If we manage to do it in time, we will have a fully functioning commerce site by the end of the workshop! Even if you didn't get to finish everything during the workshop, you are welcome to continue after the workshop and let us know how the experience went!

ℹ️ The template is mostly implemented (design-wise), but we will have to implement the logic that retrieves our data from Wix and use the Wix SDK to do so.

Technology Stack

For this workshop and our project template, the website will be based on all of the latest and greatest:

- React Server Components and Server Actions - minimum amount of Javascript required, all operations are done on the Server.
- The template is designed with TailwindCSS.
- We'll be using the Wix SDK and our different business modules.
- Typescript
- Deployment on Vercel - deploy on every commit, fast global deployment with minimum latency and advanced caching

💡 Check out our full and final result at: https://wix-nextjs-commerce.vercel.app

Getting Started

Prerequisites

Before getting started make sure that you have the following installed:

- Node v18 (you can use fnm or nvm to install the correct version)
- Pnpm (`npm i -g pnpm` - https://pnpm.io/)

Cloning the Project Locally

To get started go ahead and fork our base template to your Github account:
https://github.com/wix-playground/commerce-workshop

This template uses `pnpm` as the package manager, so after cloning go ahead and install:

Unset
```
pnpm install
```

Now you are ready to start developing. To see our (not yet working) commerce site in action, run:

Unset
```
pnpm dev
```

At this point you'll see an empty home page, and much of the functionality will not work.

Setup your Wix Headless Project

Before we start implementing code in our project, we first need to set up a Wix Headless project where we will manage our store, products, collections, CMS and more.
To get started creating your first Wix Headless project, navigate to
https://www.wix.com/developers/headless and follow the instructions to create your project.

❗ Be sure to select Wix Stores as one of the business solutions you want to install on your new Wix Headless project

ℹ️ Need some help and guidance? You can follow along the documentation in this article https://dev.wix.com/docs/go-headless/getting-started/setup/general-setup/overview
Note that we will be using OAuth for our project authentication, so make sure to follow the instruction for creating an OAuth app

Preparing your Wix Store for development

Our commerce site is going to display all of the products that we have on our store, and will allow visitors to purchase them with an online checkout. Apart from displaying all the products, our products will be managed in Product Collections - named groups of products that belong together. The products that will be displayed on our website homepage will be retrieved from special collections with specific names that are known in advance.

Go ahead and create the following collections (Categories as they are called in the Wix Dashboard):

- `hidden-homepage-carousel` - The products you add to this collection will be displayed in a carousel on the bottom of the homepage.
- `hidden-homepage-featured-items` - These products will be displayed as featured products above the fold in the website homepage.

💡 You can create more collections (like `Clothes`, `House Items`, etc) that will also be displayed on our site in the search results page.

💡 Need some example products to display on your site? We've got you covered. Download this csv file and import it to your store.

Connecting the Local Project to Wix

For our local project to be able to start making requests to our Wix Store and retrieve data, we need to set up an environment variable with our Wix Client ID - which is used for our authentication. From the OAuth application you created in Setup your Wix Headless Project copy the Client ID, and create a `.env.local` file at the root of your project with the following content:

```
Unset
COMPANY_NAME="Vercel Inc."
```

```
TWITTER_CREATOR="@vercel"
TWITTER_SITE="https://nextjs.org/commerce"
SITE_NAME="Next.js Commerce"
WIX_CLIENT_ID="<your Client ID here>"
```

❗ Do not forget to replace `<your Client ID here>` with the actual client ID you copied from your headless settings page.

You are ready to start implementing the code for our website! 🚀

Optional: Set up your Project on Vercel

If you want to deploy your website on a real URL that will be accessible to other collaborators, you can easily set up your new project on Vercel. In each step of this workshop, you can always commit and push your changes, and Vercel will automatically deploy your new code to production. This will allow you to always be in production and experience the continuous deployment approach of Vercel.

To set up your project on Vercel, go ahead and run the Vercel CLI and follow the instructions there. The CLI will set you up with an account, create a new project and deploy your first version.

```
Unset
npm install -g vercel
vercel deploy
```

❗ For your production deployment to work correctly as well, you'll need to setup the same `NEXT_PUBLIC_WIX_CLIENT_ID` environment variable. Check out the Vercel Environment Variables documentation for more information.

Implementation Guidelines

💡 The full solution can be seen at https://github.com/wix-playground/commerce-workshop/pull/1. Feel free to take a look if you get stuck or need a small nudge in the right direction.

We will implement our project on a step-by-step basis. There is no rush to finish all of the steps - take your time and make sure you understand the code you implemented. You can always continue after the Workshop, or come ask us for guidance and we will be happy to help.

All of the code that we will need to implement exists in 2 files in the project:

- `lib/wix/index.ts` - All of our functions that retrieve data exist there, and we will need to implement the function bodies.
- `middleware.ts` - The logic for ensuring a valid visitor session exists there.

Setting up the Wix SDK

Our SDK and SDK business modules are already installed in the project and can be found in the `package.json`:

- `@wix/sdk` - The main SDK package where we can create a client. Docs
- `@wix/ecom` - The business module for basic commerce logic - cart, recommendations and more. Docs
- `@wix/stores` - The business module for managing a store - products, collections. Docs
- `@wix/redirects` -The module for managing redirect sessions to Wix managed pages. Docs
- `@wix/data` - The module for working with Wix Data Collection for querying, inserting, etc. Docs

📚 Be sure to read the documentation of the core SDK module to familiarize yourself with the way to make API calls. You'll find the WixClient.use function especially useful for the implementation in this workshop.

Implementing the Search Results Page

Now we can start making requests to Wix to start retrieving products, collections and more. The first part of our website that we will implement is the Search Results page. If you go to the homepage right now, type in something in the search box, you'll see that you are always getting nothing, even if you do have products in your Wix store.

🎯 Implement the `getProducts` function in `lib/wix/index.ts`. See the instructions in the function body.

💡 The app code uses some types to represent Product that don't match exactly to the Product type that is returned from the Wix API. Make sure to **use the `reshapeProduct` function** to map the Wix product to the product structure that the UI expects.

📚 Documentation for working with the products API namespace can be found at https://dev.wix.com/docs/sdk/api-reference/stores/products/introduction

Implementing the Product Page

Now that we have a search results page working, when clicking on a specific product, you'll notice that you are getting a 404 error. We need to implement the function to get a specific product.

🎯 Implement the `getProduct` function in `lib/wix/index.ts`. See the instructions in the function body.

Bonus: Implementing the Product Recommendations

Now we have a working Product Page and we can see the product details. But at the bottom of the Product page there is a section called Related Products that shows other products that the store visitor might be interested in. For that we will need to implement a function that returns product recommendations based on a product id.

🎯 Implement the `getProductRecommendations` function in `lib/wix/index.ts`. See the instructions in the function body.

Implementing the Collections section in the Search page

In the search results page, you'll notice that there is a collections section at the left sidebar. But currently it only shows one hard coded collection. We want to display the collections that we have previously created. For that we will need to implement a function that returns all the collections on our store.

🎯 Implement the `getCollections` function in `lib/wix/index.ts`. See the instructions in the function body.

💡 The app code uses some types to represent Collection that don't match exactly to the Collection type that is returned from the Wix api. Make sure to use the `reshapeCollection` function to map the Wix collection to the collection structure that the UI expects.

📚 Documentation for working with the collections API namespace can be found at https://dev.wix.com/docs/sdk/api-reference/stores/collections/setup

Now we can see our collections in the left sidebar, but if we go to a specific collection, we will get a 404 page. For that we need to implement a function that returns data about a specific collection.

🎯 Implement the `getCollection` function in `lib/wix/index.ts`. See the instructions in the function body.

Now that we have this function implemented when we go to a collection page, we no longer get a 404 page, but we don't get any products in the collection. For that, we will need to implement a function that returns the products in a specific collection.

🎯 Implement the `getCollectionProducts` function in `lib/wix/index.ts`. See the instructions in the function body.

We now have the full collections and products functionality working! If you go back to the homepage, you'll see that the homepage also presents products now! This is because the code can now retrieve the hidden collections we created during the set up.

Implementing the Cart Functionality

To complete the experience on our commerce site, we need to finish implementing the cart functionality and allow users to perform an online checkout with the selected products.

Setting up Visitor Sessions

When making requests to Wix, we have to make those with a valid identity. Since we are building a website where anyone can browse the site, we are going to need to create Visitor identities. You can read the full documentation on how to set up a visitor session at Handing Visitors.
Up until now, we didn't need to do anything explicit to work with a visitor identity. Behind the scenes, the Wix SDK automatically created visitor authentication tokens for us, and used those to make the requests. But when we come to implement the cart functionality, it becomes important for us to store the visitor token in a persistent way. That is because the Wix currentCart functionality works by identifying the visitor id that is making the requests, and so Wix knows how to return the correct cart, add to the correct cart, etc. Without storing the visitor tokens between refreshes, everytime a user refreshes the browser their cart contents will disappear! Obviously this is not the wanted behavior, and so we will implement the tokens persistence in the browser cookies (what we call a Visitor Session).

Implementing `middleware.ts`

NextJS offers a way to create a middleware, which is a function that will run on every request made to our site. This is the perfect place for us to place the logic that ensures we have a valid visitor session cookie, and create the cookie if it doesn't exist. Thus the rest of the code can be sure that the cookie exists and we don't have to handle it in other parts of our app.

In our `middleware.ts` we need to implement the following logic:

- Read the cookies from the request and look for the `WIX_REFRESH_TOKEN_COOKIE` cookie. (you can read the cookies from `request.cookies`).
- If the cookie exists, we can let the request path through to the next handler. (just return NextResponse.next()).
- If the cookie doesn't exist, we will need to create a new visitor (create a client and call `generateVisitorTokens;), and with its tokens, create the `WIX_REFRESH_TOKEN_COOKIE` cookie (on the NextResponse object we can set cookies with res.cookies.set), store it on the response and then continue to the handler.

Implementing `getWixClient` in `lib/wix/index.ts`

Now that we ensured that we have a cookie with a valid visitor refresh token, we can create a `WixClient` in our handlers (`lib/wix/index.ts`) that will have the refresh token stored inside, and so all requests that will be made with the `WixClient` will have the correct identity.
Go ahead and change the implementation of `getWixClient` function in `lib/wix/index.ts`, so that it reads the `WIX_REFRESH_TOKEN_COOKIE` from the cookies (`import {cookies} from 'next/headers'`) and then passes the parsed tokens to OAuthStrategy).

Implementing currentCart Functionality

If you go to a specific product page, you'll notice that clicking Add to Cart results in an error message. For that we need to implement the add to cart functionality.

🎯 Implement the `addToCart` function in `lib/wix/index.ts`. See the instructions in the function body.

💡 The app code uses some types to represent Cart that don't match exactly to the Cart type that is returned from the Wix api. Make sure to use the `reshapeCart` function to map the Wix cart to the cart structure that the UI expects.

📚 Documentation for working with the currentCart API namespace can be found at https://dev.wix.com/docs/sdk/api-reference/ecom/current-cart/setup

Now the Add to Cart button no longer throws an error, but nothing is added to the cart. That makes sense because we haven't implemented the logic to retrieve the contents of the cart. We will do that now.

🎯 Implement the `getCart` function in `lib/wix/index.ts`. See the instructions in the function body.

Now we can see the products that we have added to the cart! To finish our cart functionality, we need to allow our visitors to remove items from the cart and to change the quantity of the items in the cart.

🎯 Implement the `removeFromCart` function in `lib/wix/index.ts`. See the instructions in the function body.

🎯 Implement the `updateCart` function in `lib/wix/index.ts`. See the instructions in the function body.

Our cart is now fully functional! The last thing we need to allow our users is to checkout with the contents of their cart.

Implementing the Cart Checkout

To perform the checkout, we are not going to implement the checkout page (accepting payments is extremely hard!) but we are going to use the Wix managed pages and Wix Redirect Sessions. You can read more about implementing the Checkout flow in the documentation.

🎯 Implement the `createCheckoutUrl` function in `lib/wix/index.ts`. See the instructions in the function body.

Now you can try to checkout and you'll see that you are redirected to a Wix Checkout page and you can purchase the products in your cart and make that $$$!!

Bonus: Implementing the CMS logic

For any commerce site, just displaying products and collections and allowing people to purchase them is not enough. A store needs to display more information about the store, like information about the store itself, its return policy and more. For that a commerce site usually needs to integrate with a CMS system, so that a store owner can easily change the contents of those pages without requiring the developer to make changes to the site (developers aren't cheap you know 😀).

Our NextJS Commerce uses a CMS system for 2 parts: Defining the menus dynamically (in the header and in the footer) and defining the pages and their contents dynamically. This allows the store owner to change the menu, add and remove pages at their will. Since Wix is a great CMS itself, we will be using Wix as a CMS as well.

Modeling the Wix Data Collections

Before we start implementing the missing code to make the menus and pages work, we need to first model our menus and pages in Wix CMS. For that we will create 2 Wix Data Collections on our headless project: one for the menus (header and footer) and one for the pages and their content.

💡 Checkout the code for `getMenu`, `getPage` and `getPages` in `lib/wix/index.ts` and the types for `Page` and `Menu` and think how to best model your Wix Data Collections to it will be easier for you to implement the needed logic in those functions

After you have created and modeled your Wix Data Collections, go ahead and insert some data into your new collections.

- Make sure to have 2 menus with the following handles (in whatever way you modeled your data). The code in the template is going to request the menu with these handles: `next-js-frontend-header-menu` and `next-js-frontend-footer-menu`.
- Create a few pages you can place in your menus. You can create any pages you like. Some ideas are: an About page, Terms and Conditions, Shipping and Return Policy, FAQ and more.

Implement the Missing CMS functionality

First we'll want our menu to appear on the header and the footer so that we can easily navigate to those pages. For that we'll need to implement a function that returns the menu items based on the menu handle.

🎯 Implement the `getMenu` function in `lib/wix/index.ts`. See the instructions in the function body.

Now we can see our menu in the header and the footer, but if we navigate to one of the pages, we'll see that we don't see anything on the page. For that we need to implement a function that returns the page data based on the page path.

🎯 Implement the `getPage` function in `lib/wix/index.ts`. See the instructions in the function body.

Now when we navigate to one of the pages in the menu we can actually see the contents of the page! 🥂

The very last part of our implementation has to do with the website sitemap.xml. NextJS allows you to easily create a sitemap.xml that maps all of the pages that exist on a site which is very helpful for SEO and web crawlers to understand the structure of a website. To make our sitemap.xml work correctly and display our dynamic pages from the CMS, we'll need to implement a function that returns all the dynamic pages.

🎯 Implement the `getPages` function in `lib/wix/index.ts`. See the instructions in the function body.

That's it! Hooray, you made it all the way to the end of the workshop! 🎈

Summary

We are very happy that you have participated in our Wix Headless workshop! We are looking forward to seeing what else you can use Wix Headless for, the future is full of possibilities!
We would love to hear your thoughts and feedback on anything related to Wix Headless, the workshop, our APIs & SDKs and anything else you would like to share with us. Your feedback is valuable and will help us improve the product for all Wix users!
You can always find us at:

- Headless Discord Channel - This is where we interact publicly with Wix users. You are welcome to join, check out the community and help us build a strong and friendly community.
- contact-headless@wix.com - If you are into email, that’s ok too, everyone has their preferences…