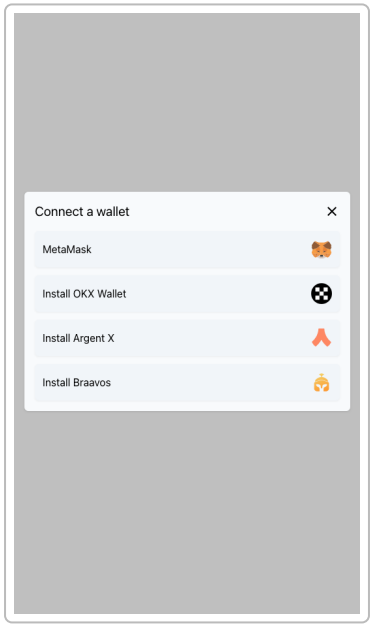
ChatGPT

UI Technologies for StarkNet DeFi & Marketplace Applications

Building user-friendly decentralized finance (DeFi) and marketplace dApps on **StarkNet** involves a stack of modern UI frameworks, libraries, and tools that bridge the front-end to Cairo smart contracts. Below is a structured overview of the most widely used technologies in this ecosystem, including wallet integrations, contract interaction libraries, front-end frameworks, developer SDKs, UI components, and best practices for responsive, secure, and performant StarkNet dApps.

StarkNet Wallet Integration (Argent X, Braavos, etc.)



Example of a wallet selection modal in a StarkNet dApp. Users can choose among StarkNet-compatible wallets (e.g. MetaMask with StarkNet Snap, OKX, Argent X, Braavos) to connect [1] .

StarkNet uses an **account abstraction** model where user wallets are smart contracts. The two most popular wallets are **Argent X** and **Braavos**, which together cover nearly the entire user base (about 70% Argent X vs 30% Braavos) [2] . Both are non-custodial, open-source smart contract wallets with advanced features like **Multicall** (batching multiple actions into one transaction) and optional **2FA security** [3] . Developers typically integrate these wallets via browser extensions (Argent X, Braavos extension) or mobile apps (Braavos mobile), enabling users to approve transactions with their StarkNet accounts.

To facilitate wallet connectivity in the UI, developers use dedicated **wallet adapter libraries**. The standard is **get-starknet**, a community SDK that detects installed StarkNet wallets and provides a unified API to request connections [4] . Using get-starknet, a dApp can present a modal listing all available wallets and

retrieve a **StarkNet Window Object (SWO)** for the chosen wallet [5]. This allows the dApp to access the wallet's account, address, and signing capabilities in the browser. For example, when the user clicks "Connect Wallet", get-starknet can display a popup with options (Argent X, Braavos, etc.) and then return the connected wallet object for signing transactions [1].

An evolution of get-starknet is **StarknetKit**, an open-source SDK from Argent. StarknetKit streamlines StarkNet wallet integration with one-line setup and greater customization [6]. It lets developers *choose which wallets* to show in the modal (and in what order) and even add custom wallet connectors, thanks to a modular design inspired by StarkNet React hooks [7]. In practice, StarknetKit can be installed via NPM (`yarn add starknetkit`) and provides simple methods like `connect()` to trigger wallet selection and connection [8]. This flexibility is useful for tailoring the UX – for instance, a DeFi app might prioritize showing certain wallets first or integrate a new wallet as the ecosystem grows.

StarkNet React (discussed more below) also includes built-in *connectors* for injected wallets. With Starknet React, you can import predefined connectors like `argent()` and `braavos()` and include them in your app's provider configuration [9]. This means React dApps can support Argent X and Braavos out-of-the-box by leveraging those connectors, which handle the details of interfacing with the extension. There is also emerging support for StarkNet in MetaMask via a **StarkNet Snap** – MetaMask's developer docs show that get-starknet can detect MetaMask and prompt users to enable the StarkNet Snap plugin [1]. Other wallets like **OKX Wallet** are entering the StarkNet space as well (as seen in the modal above). In summary, for wallet integration on StarkNet, developers rely on Argent X and Braavos as the primary wallets, and use connector libraries (get-starknet, StarknetKit, or Starknet React hooks) to provide a smooth "connect wallet" flow in the UI.

Smart Contract Interaction with Cairo (StarkNet.js)

Once a wallet is connected, the front-end needs to interact with Cairo smart contracts deployed on StarkNet. The primary library for this is **StarkNet.js**, an official JavaScript/TypeScript SDK for StarkNet. StarkNet.js allows developers to call contract functions, send transactions, and query on-chain data from a web application [10]. It is analogous to Ethereum's Ethers.js or Web3.js, but tailored for StarkNet's architecture. In fact, StarkNet.js has become *"the most popular library for interacting with StarkNet"* for front-end developers [10]. It is open-source and continuously updated alongside StarkNet's network upgrades.

Using StarkNet.js, you typically work with a **Provider**, **Signer**, and **Contract** abstraction. A **Provider** represents a connection to a StarkNet node (RPC endpoint) for reading blockchain state, while a **Signer** (usually provided by the user's wallet) is used to sign transactions for state-changing operations [11]. This separation is a best practice for efficiency: read-only calls (e.g. fetching a DeFi pool's TVL or an NFT's metadata) can be done through a provider without involving the user's private key, whereas writes (e.g. swapping tokens, listing an NFT) require the Signer to authorize. StarkNet.js makes it easy to configure a provider (for example, pointing to a service like an Alchemy or Infura StarkNet RPC) and then use the user's connected wallet as the signer for transactions. Under the hood, **Argent X and Braavos inject a Signer** object when connected, which StarkNet.js can utilize via its **WalletAccount** class [12] [13]. The WalletAccount abstraction in StarkNet.js is specifically designed to forward signing requests to the browser wallet and send transactions through it [14] [15].

Working with a contract in StarkNet.js involves obtaining the contract's **ABI** (application binary interface) and deployed **address**, then creating a `Contract` instance. The ABI is a JSON describing the contract's

functions and data structures (similar to ABI in Ethereum), and it's needed for StarkNet.js to encode function calls. Once you have `const myContract = new Contract(abi, address, providerOrAccount)`, you can call read-only methods or invoke transactions. For example, you might call `await myContract.get_balance(userAddress)` or send a transaction `await myContract.transfer(recipient, amount, {from: myWalletAccount})`. Under the hood, if you provided a `WalletAccount` (connected wallet) as the account, StarkNet.js will prompt the user's wallet to sign the transaction and then send it to the network.

It's worth noting that StarkNet's account model means **every user action is a contract call**. Even the user's own account is a contract that must exist on-chain. A new StarkNet account (wallet) is not usable until it's **deployed and funded**. Unlike MetaMask's EOA accounts that exist by default, deploying a StarkNet account incurs an on-chain transaction (often handled behind the scenes when the user creates a wallet) [16]. As a developer, you should account for this in the UI: for example, a fresh user may need to activate their wallet (deploy account contract) and deposit ETH for fees before they can interact with other contracts. Good dApp UIs will detect if the user's account is not yet deployed or lacks ETH and guide them (perhaps by linking to the StarkNet faucet for testnet, or explaining the one-time deployment step) [17].

In summary, **StarkNet.js** is the go-to library for front-end <-> Cairo contract interaction. It provides the low-level tools to query contract state and send transactions. In addition, similar SDKs exist in other languages (e.g. **starknet.py** for Python backend scripts, **starknet-rs** for Rust, etc.), but for a web UI you will almost always use the JavaScript/TypeScript library. Community tutorials and docs emphasize setting up StarkNet.js with your contract's ABI and leveraging providers (from services like Infura, Alchemy, Moralis, Chainstack, etc.) to handle JSON-RPC calls [18]. With these tools, a DeFi frontend can read pool balances or interest rates from Cairo contracts and trigger functions like swaps or deposits, while an NFT marketplace UI can fetch token metadata and invoke purchase or listing functions – all through the StarkNet.js interface.

Front-End Frameworks (React, Vue, etc.) for StarkNet

You can build a StarkNet dApp front-end with any modern web framework, but **React** is by far the most widely used in the blockchain dApp community. React's component model and rich ecosystem (Next.js, create-react-app, Vite, etc.) make it a natural choice. In fact, many StarkNet example projects and boilerplates use React or Next.js. For instance, StarkWare's official tutorial uses React for the UI, but notes that you could just as well use **Next.js, Vite, or others** – the choice of framework does not limit StarkNet integration [19]. React offers familiarity and a large pool of UI libraries, which helps developers build complex interfaces (like DeFi dashboards) quickly.

Next.js (a React framework) is commonly used for dApps due to its routing and performance features. **Vue.js** is also an option – while not as prevalent as React in web3, it's certainly capable of StarkNet integration. There is a community boilerplate called **vue-stark-boil** that demonstrates connecting a Vue 3 application to StarkNet [20]. Similarly, some developers have created **Svelte** examples: for instance, a SvelteKit template that uses StarknetKit to connect to a Cairo counter contract [21]. These examples show that StarkNet integration is framework-agnostic – as long as you can call JavaScript libraries (like starknet.js or starknet.js-based connectors), you can use React, Vue, Svelte, Angular, or others per your preference.

That said, React enjoys the richest ecosystem of StarkNet-specific tooling. **Starknet React** is a library of React hooks that greatly simplify working with StarkNet in React apps. It provides hooks for wallet

connections, contract calls, and state management that fit naturally into React's paradigm. We'll discuss Starknet React in the next section, but its existence has made React even more attractive for StarkNet dApps because it reduces boilerplate. Additionally, many StarkNet community projects (including the Scaffold projects mentioned below) are written in React/Next.js, which means if you're using those as a starting point, you'll be in the React world by default.

In summary, **React/Next.js** is the top choice for building StarkNet UIs (with TypeScript recommended for safety), while **Vue** and **Svelte** are also viable with community support. The key is to choose a framework you're comfortable with, and ensure it can integrate the StarkNet JS SDK or connectors. Most frameworks allow using NPM packages and global window objects, so connecting to wallets and calling contracts works similarly across frameworks. Just be mindful of build setups for things like polyfills (for example, StarkNet.js may require polyfills for Node modules in some bundlers – using Vite or Next with proper config can handle this).

Developer Tooling and SDKs (Starknet.js, Starknet React, etc.)

Beyond the base StarkNet.js library, there are higher-level SDKs and tools that make life easier for developers building UIs:

- **Starknet.js** – As described, the core JS library for StarkNet. It provides classes for Provider, Account/Wallet, Contract, Signer, etc. It's open-source and constantly updated alongside protocol changes [10]. If you're doing anything in a web app with StarkNet, this is likely in your `package.json`. It supports both promise-based and async/await usage for contract calls and transactions.

- **Starknet React** – A powerful collection of React **hooks** and utilities specifically for StarkNet dApps. Starknet React combines StarkNet.js under the hood with state management tools to offer a declarative, React-style interface [22]. For example, it provides hooks like `useContract` and `useReadContract` to bind to contract calls, `useAccount` to get the connected wallet account, `useBalance` to fetch token balances, `useSendTransaction` to send transactions, etc. Starknet React integrates **TanStack Query** (formerly React Query) for caching and updating data, and **abi-wan-kanabi** (a type-safe ABI TypeScript library) to give you typed contract interactions [22]. This means you get features like caching of contract reads, automatic refresh on new blocks, and TypeScript types for your contract's functions – all of which improve developer experience and UI responsiveness. Using Starknet React, a lot of boilerplate (like manually polling for new blocks or handling loading states) is handled for you. For instance, there's a `useInvalidateOnBlock` helper that will refetch certain queries whenever a new block is mined [23], which is great for keeping UI data (like DeFi pool stats) live-updated without excessive manual polling.

- **Wallet Connector SDKs** – We discussed **get-starknet** and **StarknetKit** in the wallet section. These are also part of the developer toolkit. They abstract differences between wallet implementations and provide a consistent API to connect/disconnect wallets and sign transactions. StarknetKit in particular is an SDK that builds on starknet.js to simplify wallet onboarding in apps [6]. Similarly, **starknet-react** includes ready connectors for common wallets [24] as shown earlier.

- **Scaffold Projects** – The StarkNet community has produced some scaffold/template repositories that bundle best practices and tools. **Scaffold-Stark** (often called StarkNet Scaffold) is one such toolkit,

analogous to Ethereum's Scaffold-ETH. It's a forkable, production-grade dApp template using Next.js, Starknet.js, Starknet React, and StarkNet's own dev tools [25]. By doing `npx create-starknet-app`, developers can generate a new project that has a working example of contract integration and UI components. These scaffolds often include scripts for running a local StarkNet devnet, deploying example Cairo contracts, and a frontend with wallet connection and sample transactions. For example, *Horus Labs' Starknet-Scaffold* comes with default **wallet support for all StarkNet wallets**, preconfigured via StarknetKit, and has UI components for common needs like a burner account (temporary development wallet), faucet integration, and more [26] [27]. Using such a scaffold can drastically speed up development, as you get a working StarkNet dApp framework out of the box.

- **IDE and Debugging Tools** – While not UI libraries per se, it's worth noting tools like **Starkli** (the StarkNet CLI) and **StarkNet Devnet** for local testing. Starkli is a command-line tool for StarkNet (much like Hardhat or Truffle for Ethereum) [28], and StarkNet Devnet is a local network simulator. For UI developers, having a local devnet means you can test contract interactions without needing an internet connection or testnet faucet, which accelerates development and debugging. You'd deploy your Cairo contracts to devnet and point your Starknet.js provider to the devnet's RPC URL. This way, you can iterate on UI features (like a new DeFi pool interface or NFT listing flow) using dummy data on a local chain, and only switch to testnet for final testing.

- **Other SDKs** – If your dApp requires backend services or indexing, StarkNet also has tooling for that. **Starknet.py** (Python SDK) allows writing backend scripts or services that interact with StarkNet (for example, a server to aggregate stats or to listen to events). **The Graph** is not yet fully integrated with StarkNet at the time of writing, but projects like **Snapback** and **ApeX** have used custom indexers. Additionally, **NFT APIs** (e.g. NFTScan for StarkNet) can serve NFT metadata and collection info to your UI without you parsing it directly from chain [29] [30]. These are more on the data layer side, but worth considering if you need to display complex data (like historical charts or large NFT galleries) efficiently.

In summary, the developer tooling around StarkNet UI is growing rapidly. Starknet.js and Starknet React form the core of interacting with contracts and managing state. Surrounding that, wallet SDKs (get-starknet, StarknetKit), scaffold apps, and various language-specific libraries provide a robust toolkit. All of these are open-source and community-supported, with active development as StarkNet matures. Checking StarkNet's official documentation and community "awesome StarkNet" repositories [25] [20] is a good way to keep track of new tools and libraries as they appear.

Web3 UI Components and Design Frameworks

Designing a polished UI for a DeFi protocol or NFT marketplace often means using existing UI component libraries and following web3-specific UI patterns. While StarkNet doesn't have a **must-use** UI kit exclusive to it, developers leverage both general-purpose component libraries and web3-tailored components:

- **Reusable Web3 Components**: Many dApps implement similar UI elements – wallet connect buttons, network switchers, address displays, token icons, transaction status modals, etc. Instead of reinventing the wheel, you can use libraries like **Web3UIkit** (by Moralis). Web3UIkit is an open-source set of "beautiful and lightweight UI components for web3 developers" [31]. It includes React components for common needs like Connect Wallet modals, notification toasts for transactions, address blockies (visual hash icons), and even pre-styled components for displaying NFTs,

cryptocurrency amounts, and more. This kit is chain-agnostic in many parts, though some components are EVM-centric. You could integrate it with StarkNet by using the generic components (for layout, modals, buttons, etc.) and wiring them to StarkNet-specific logic. For example, Web3UIkit has a `<CryptoLogos chain="ETH" />` component; StarkNet being an L2 would require a custom logo or using an Ethereum logo if appropriate. The key benefit is speeding up development with a cohesive, tested UI style.

- **StarkNet Scaffold Components**: As mentioned earlier, projects like Starknet-Scaffold provide ready-made components that are tailored to StarkNet. For instance, the scaffold includes an **AddressBar** component that displays the user's StarkNet address, with an identicon (using the `react-blockies` library to generate a unique avatar for the address) and options to copy the address or disconnect the wallet [32]. It also has a **NetworkSwitcher** component for toggling between StarkNet mainnet and testnet in the UI [33], a **Transactions** history modal, and an **AccountBalance** display for showing the user's ETH or token balances [34]. These are all common UX elements in blockchain apps. By studying or borrowing from these open-source components, you ensure your dApp provides the expected features (like the ability to switch networks or quickly copy one's address) with consistent design. Many of these components use simple libraries such as blockies (for avatars) or interact with StarkNet React hooks (e.g., useContractRead for balances [32]) under the hood.

- **UI Frameworks (CSS and Components)**: In terms of styling and layout, developers often use popular frameworks like **Chakra UI**, **Material-UI**, **Ant Design**, or **Tailwind CSS** to build dApp interfaces. These aren't specific to web3, but they provide accessible and responsive components (grids, forms, modals, buttons, etc.) that can be themed for a dark/light mode typical of DeFi apps. For example, the official starknet.io website is built with Chakra UI and Framer Motion [35], which shows that Chakra's components work well even for StarkNet-related sites. **Tailwind CSS** is also common in many crypto projects for quickly styling components with utility classes, ensuring consistency across the app. The choice here usually comes down to developer preference. What's important is to maintain a clean UI/UX: clear typography, a responsive layout for desktop and mobile (since users may interact via mobile wallets or wallet browser extensions on mobile), and theming that matches the project's brand.

- **Web3 Patterns**: Certain UI patterns are specific to blockchain applications. For instance, **notification toasts** or pop-ups indicating transaction status (submitted, pending, confirmed, failed) are crucial for good UX, as blockchain transactions are asynchronous. Libraries like **react-hot-toast** or **web3-react's notifications** can be used, or the StarkNet React hook `useTransactionReceipt` to poll transaction status and then display a message. Another pattern is showing **identifiers in short form** (e.g. an address like `0xabc...def` with a copy button) – many UI kits or simple utility functions handle truncating and copying addresses. Ensuring your dApp has these little quality-of-life features will make it feel much more polished and "web3-friendly" to users.

- **NFT and DeFi specific components**: If you're building an NFT marketplace UI, you might incorporate components for gallery displays, image lazy loading (to handle potentially large images from IPFS), and even 3D/AR viewers for NFTs. Some libraries exist for viewing NFTs (for example, thirdweb's React SDK has some NFT renderer components, though StarkNet support for thirdweb is pending [36]). You might use a generic carousel or gallery library and tie it into StarkNet data. For DeFi dashboards, you might integrate charting libraries (like Recharts or Chart.js) to display historical

data, or data grid libraries for order books and trade history. While these are not StarkNet-specific, they form part of the overall tech stack of a dApp UI.

In essence, **UI kits and components** for StarkNet dApps are a mix of general web UI libraries and web3-specific packages. It's wise to leverage open-source components that the community has battle-tested, especially for critical features like wallet connections and transaction notifications. By doing so, you not only speed up development but also adhere to familiar design patterns that users expect (for example, a consistent way to connect their wallet or view their address). Always keep an eye on StarkNet community resources – as the ecosystem grows, more StarkNet-tailored UI components may emerge (similar to how Ethereum has things like RainbowKit or Wagmi for EVM chains). Currently, the combination of Starknet React + a component library (Chakra/MUI) + some custom web3 components covers most needs for building a slick DeFi or NFT app UI on StarkNet.

Best Practices for Responsive, Secure, and Performant StarkNet UIs

When building a front-end for a StarkNet dApp, developers should follow best practices both from general web development and web3-specific considerations:

- **Responsive Design**: Ensure your application works well on various devices and screen sizes. Many StarkNet users might interact via desktop (browser extension wallets) but others could use mobile wallets or wallet browsers. Use responsive CSS frameworks or media queries to adapt layouts. Test your dApp on mobile resolutions; for instance, Braavos mobile wallet has an in-app browser where users could load your dApp. A responsive UI with a flexible grid and scalable typography will improve overall user experience.

- **Performance Optimization**: StarkNet is a Layer 2 with fast finality compared to L1, but there can still be delays (a few seconds) for transactions. Optimize your UI to handle these gracefully. This means providing immediate feedback when a user initiates an action – e.g., disable the submit button and show a "Transaction submitted…" state. Use loading spinners or skeleton components for data that's being fetched (like updating DeFi pool stats). Leverage caching provided by tools like TanStack Query (as used in Starknet React) to avoid redundant network calls. For example, if your app needs to display a user's token balance in multiple places, fetch it once and reuse the data rather than calling the chain repeatedly. Utilize **multicall** where possible (Argent X and Braavos wallets support aggregating multiple contract calls in one transaction [37]), and consider batching read calls using `callArray` with StarkNet's JSON-RPC if available, to reduce latency. If your app shows real-time data (like prices or oracle feeds), consider WebSocket or periodic refresh strategies, but be mindful of not spamming the RPC – a common approach is to update on each new StarkNet block (which Starknet React's `useInvalidateOnBlock` can help with). Also, offload heavy computations from the front-end: for example, if you need to aggregate data or filter a large list of NFTs, see if it can be done via an indexer or backend, and send the front-end only the needed results.

- **Security Best Practices**: Front-end security in a blockchain context is crucial. Always **validate user inputs** before sending them to a contract call – for instance, if a user is entering an amount of tokens to swap, ensure it's a positive number and within their balance. Use form libraries or HTML5 validation for this, and double-check in your contract (but providing immediate UI validation prevents many mistakes). Be careful with **transaction composition**: if your UI constructs calldata for

Cairo contracts, use the official starknet.js methods or a well-tested ABI encoding library rather than hand-crafting bytes. This avoids encoding bugs that could lead to failed transactions or vulnerabilities. **Phishing and Signature Safety**: When prompting the user to sign a message or transaction, clearly explain in the UI what they are approving. StarkNet's wallets will show the contract address and entry point being called; ensure your dApp provides context (e.g., "You are about to swap X for Y on AMM Z"). This builds trust and helps users avoid malicious transactions. Also, never ask the user for their private key or seed phrase – all signing should occur in the wallet. If you integrate with a web wallet or custom key management (like Web3Auth or similar social login wallets in future), be sure that keys are stored securely (usually in the browser's isolated storage or not at all on the front-end).

- **Account Abstraction UX**: As mentioned, StarkNet accounts are contracts. A best practice is to **abstract the complexity** of this from the user when possible. For example, if a user must deploy their account contract, your UI should initiate that seamlessly (the wallet usually guides them). If your dApp is likely to onboard completely new users, consider linking to educational material about creating a StarkNet wallet. Some dApps even build an onboarding modal that detects no wallet and offers to create one via Argent's web wallet or directs to install Braavos. Additionally, account abstraction allows features like social recovery or sponsored transactions – if your dApp plans to offer gasless transactions (paying fees on behalf of user), ensure the UI communicates that clearly ("Gas fees sponsored by dApp for this action"). Always test the **first-time user flow**: a user with no StarkNet account at all – see how your dApp behaves and make improvements so that they don't drop off at the first hurdle.

- **Use of Community Audited Libraries**: Stick to the well-known libraries for critical functions. For example, use starknet.js for signing and Starknet React for managing state instead of a custom homemade solution. These libraries are open-source and have community eyes on them. If you use additional packages (say a QR code library for walletconnect, or a date library for timestamps), keep them updated to pull in security fixes. Regularly audit your dependencies for any known vulnerabilities (tools like `npm audit` can help).

- **Testing and Monitoring**: Just as you would with any web app, test your dApp thoroughly. This includes unit tests for any utility functions (perhaps using Jest), integration tests that simulate clicking buttons and ensuring the right contract calls happen (some use Cypress or Playwright to automate a browser with a test wallet). On the blockchain side, test with StarkNet's testnet (currently e.g. the Sepolia testnet for StarkNet) which your UI can be configured to point to. Monitor the app in production – if you have many users, consider using analytics or error tracking to catch UI issues. Also, be aware of StarkNet network changes or Cairo upgrades that might require updating your frontend libraries. Subscribing to StarkNet developer updates or checking the starknet.js repo periodically is wise, given the rapid development of the ecosystem.

- **User Experience & Design**: Finally, good UX principles go a long way in dApps. Keep visuals clean and information hierarchy clear (as StarkNet's own UX guide suggests, use clean layouts and minimalist design focusing on key actions) [38] . Use consistent iconography (e.g., use the StarkNet logo or ETH logo where appropriate for clarity). For marketplace apps, make sure to display relevant info (price, owner, bids) prominently. For DeFi apps, show APYs, risks, and link to documentation. A developer-friendly dApp is one that surfaces the right info for power users (like contract addresses, transaction IDs with links to a StarkNet block explorer such as Voyager or Starkscan) while still being

approachable for newcomers (with tooltips or help modals to explain concepts like L2, bridging, etc.). Security-wise, always inform users of what's happening: e.g., "Waiting for confirmation...", "Transaction on StarkNet is pending (may take ~ few seconds)", and so on. This transparency builds trust in your application.

By following these best practices, you can create a StarkNet application UI that not only leverages the best technologies and libraries, but also delivers a smooth, safe experience. The StarkNet ecosystem provides a lot of building blocks – from wallets and hooks to component kits – that you can assemble into a full-featured DeFi exchange or NFT marketplace interface. Continually engage with the StarkNet developer community (Discord, forums, GitHub) to stay updated on new tools, security announcements, and design patterns. With Cairo contracts powering the back-end and a modern front-end stack as outlined above, developers have an exciting and robust environment for building the next generation of decentralized applications on StarkNet.

References and Resources

- **Argent X & Braavos Wallets** – Official StarkNet smart contract wallets (browser extension and mobile). See Argent's comparison of StarkNet wallets [2] [39]. Both wallets are open-source and support features like multisig, social recovery, and multicall.
- **StarkNet.js Library** – JavaScript/TypeScript SDK for interacting with StarkNet contracts [10]. Documentation: *https://starknetjs.com*; see Medium guide for an introduction [40] [11].
- **get-starknet** – Wallet connection helper that bridges dApps and StarkNet wallets [4]. Provides `connect()` modal functionality [1]. GitHub: *https://github.com/starkware-industries/get-starknet*.
- **Starknet React** – React hooks for StarkNet dApps [22]. Official docs: *https://www.starknet-react.com/* (includes usage of connectors for Argent/Braavos [24] and many hooks for calls, transactions, etc.).
- **StarknetKit** – Argent's StarkNet connectivity SDK, allowing easy wallet modals and custom connectors [6]. NPM: `starknetkit`; Tutorial on Argent blog [41] [7].
- **StarkNet Scaffold / Scaffold-Stark** – Full-stack StarkNet dApp template (Next.js, Starknet.js, Starknet React, Cairo contracts). Provides out-of-the-box components (Connect modals, Network switcher, etc.) [42]. Repo: *https://github.com/onlydustxyz/scaffold-stark* (and Horus Labs variant).
- **Web3UIkit** – Moralis's React UI kit for web3, offering ready-made components for dApps [31]. GitHub: *https://github.com/web3ui/web3uikit*. While not StarkNet-specific, it's useful for common UI elements in any blockchain app.
- **Awesome StarkNet** – Community-curated list of StarkNet resources, libraries, and tools [20] [25]. A great reference to discover new SDKs, wallets, and development frameworks in the StarkNet ecosystem.

---

[1] [17] Connect to Starknet | MetaMask developer documentation
https://docs.metamask.io/wallet/how-to/use-non-evm-networks/starknet/connect-to-starknet/

[2] [3] [37] [39] Best Starknet Wallet: Argent X vs Braavos
https://www.argent.xyz/blog/best-starknet-wallet-argent-x-vs-braavos

[4] [10] [11] [16] [18] [19] [40] An In-depth Guide To Getting Started With Starknet.js | by Darlington Nnam | Medium
https://medium.com/@darlingtonnnam/an-in-depth-guide-to-getting-started-with-starknet-js-a55c04d0ccb7

5 12 13 14 15 WalletAccount | Starknet.js
https://starknetjs.com/docs/6.11.0/guides/walletAccount

6 7 8 41 Effortlessly connect your dapps to Starknet with StarknetKit
https://www.argent.xyz/blog/integrating-the-starknetkit-sdk-in-your-dapp

9 22 23 24 Getting Started – Starknet React
https://www.starknet-react.com/docs/getting-started

20 25 28 GitHub - keep-starknet-strange/awesome-starknet: A curated list of awesome StarkNet resources, libraries, tools and more
https://github.com/keep-starknet-strange/awesome-starknet

21 Starknet Test | Svelte Themes
https://sveltethemes.dev/KapiSolutions/starknet-test

26 27 Starknet Scaffold
https://www.starknetscaffold.xyz/

29 30 Guide: How to Develop Web3 DApp on Starknet Using NFTScan API? | by NFTScan | NFTScan | Medium
https://medium.com/nftscan/guide-how-to-develop-web3-dapp-on-starknet-using-nftscan-api-f671805553a6

31 web3ui/web3uikit: Lightweight reusable Web3 UI components for ...
https://github.com/web3ui/web3uikit

32 33 34 42 Part 2: Build and Deploy a dApp using Starknet-Scaffold | by Horus Labs | Medium
https://medium.com/@horuslabsio/part-2-build-and-deploy-a-dapp-using-starknet-scaffold-90f6c9094a07

35 The main repo for starknet.io - GitHub
https://github.com/starknet-io/starknet-website

36 Add support for Starknet integration - thirdweb
https://feedback.thirdweb.com/p/add-support-for-starknet-integration

38 The impact of UX on dApp adoption - Starknet
https://www.starknet.io/blog/impact-of-ux-on-dapp-adoption/