How to implement Particle Network's Smart Wallet-as-a-Service for AA-enabled social logins

Community member contribution Shoutout to@TABASCOatw for contributing the followingthird-party document ! Particle Network 's Wallet Abstraction services enable universal, Web2-adjacent onboarding and interactions through social logins. Its core technology,Smart Wallet-as-a-Service (WaaS) aims to onboard users into MPC-secured smart accounts supporting any chain. It also allows developers to offer an improved user experience through modular, fully customizable EOA/AA embedded wallets. Particle supports its Smart Wallet-as-a-Service through a Modular L1 powering chain abstraction, acting as a settlement layer across chains for a seamless multi-chain experience.

Arbitrum was one of the first blockchains supported by Particle Network's Smart Wallet-as-a-Service. Because of this, Particle Network has extensive support for:

- Arbitrum One, through:* EOA (non-AA social login)
-
    - SimpleAccount
-
    - Biconomy (V1 and V2)
-
    - Light Account
-
    - Cyber Account
- Arbitrum Nova, through:* EOA (non-AA social login)
-
    - SimpleAccount
-
    - Biconomy (V1 and V2)

Alongside a similar degree of support for Arbitrum Goerli and Sepolia.

Given its modular architecture, developers have the liberty to choose which of the above smart account implementations they onboard a user into after the social login process.

The user flow with Particle Network begins with social logins (using either a custom authentication or preset login methods provided by Particle Network), which leads to the generation of an EOA through MPC-TSS. This EOA is then used as a Signer for a smart account implementation that best fits the needs of the application in question (natively, this means a choice between SimpleAccount, Biconomy V1/V2, Light Account, and Cyber Account). A visualization of this process can be found below:

This document will go through the high-level process of creating a demo application on Arbitrum Sepolia using Particle Network's Smart Wallet-as-a-Service (specifically the Particle Auth Core SDK alongside Particle's AA SDK). Within the app, we'll onboard a user into a SimpleAccount instance via a social login and execute a gasless (sponsored) transaction.

Getting started

Throughout this guide, we'll be segmenting the configuration and utilization of these SDKs between two files:index.tsx andApp.tsx . Thus, we'll be building an application through a standardcreate-react-app structure. However, if you intend to use an alternative framework, such as Next.js, this same process will be largely applicable (extrapolating to your setup).

Dependencies

Before installing your dependencies, it's important to note that Particle Network's Modular Smart Wallet-as-a-Service isn't contained in one singular SDK. Rather, it refers to the combined usage of Particle Auth Core (to facilitate social logins leading to EOAs) and Particle's AA SDK (to take the EOA returned by Particle Auth Core and use it as a Signer for a smart account).

Understanding this, you'll need to install the following libraries:

- @particle-network/auth-core-modal
- , for social logins (can be substituted by@particle-network/connectkit
- for a RainbowKit-like connection modal).
- @particle-network/aa
- , to assign and interact with a smart account.
- @particle-network/chains
- , for connecting with Arbitrum Sepolia.

To do this, run one of the two following commands at the root of your project:

yarn

add @particle-network/auth-core-modal @particle-network/aa @particle-network/chains

OR

npm

install @particle-network/auth-core-modal @particle-network/aa @particle-network/chains

Setting up the Particle dashboard

Before jumping into the configuration process, you'll need to go to theParticle dashboard to retrieve three values required for your project.

When using any SDK offered by Particle Network, you'll routinely need aprojectId ,clientKey , andappId . These exist to authenticate your project and create a connection between your instance of Particle Auth and the Particle dashboard (which allows you to customize the application-embedded modals, track users, fund your Paymaster, and so on).

Once you've navigated to the Particle dashboard, follow the process below:

1. Create a new project through "Add New Project".
2. Click "Web" under "Your Apps" (if you intend to use an alternative platform, take a look at theplatform-specific guides on Particle's documentation
3.)
4. Choose a name and domain for your application (if you have yet to deploy or decide on a domain where you intend to deploy, feel free to use any filler one).
5. Copy theProject ID
6. ,Client Key
7. andApp ID
8. .

Given the nature of these values, it's recommended that you store them within.env variables, such asREACT_APP_PROJECT_ID ,REACT_APP_CLIENT_KEY , andREACT_APP_APP_ID (orNEXT_PUBLIC_{}_ID).

Configuring Particle Auth Core

As mentioned, we'll break the integration process into two components, each attached to a file within a standardcreate-react-app structure.

- index.tsx
- , our starting point In this example, the index file will be used as the source file for the configuration/initialization of Particle Auth Core.
- App.tsx
- . After setting upindex.tsx
- , we'll organize all of the core application-level logic (such as the facilitation of social login) within our App component.

Particle Auth Core is configured by one component:AuthCoreContextProvider , which takes a number of required (projectId ,clientKey ,appId) and optional values.

AuthCoreContextProvider should wrap the primary component where you intend to use Particle Auth Core. In this case, that's our App component (fromApp.tsx).

Therefore, within our index file, we'll render a constructed instance ofAuthCoreContextProvider so that we can (as the name suggests) provide context to the component wrapped within it (App).

In this example, we'll be defining the following parameters forAuthCoreContextProvider .

- projectId
- ,clientKey
- , andappId
- . We previously retrieved these from theParticle dashboard
- .
- erc4337
- , to dictate which type of smart account will be shown on the (optional) embedded wallet modal after login.* name
-
    - , the name of the smart account implementation you intend on using. In this example, we'll be using"SIMPLE"
-

    - .
-
    - version
-
    - , the version of the smart account implementation you intend on using. This should generally be1.0.0
-
    - unless you're using Biconomy V2 (in which this should be2.0.0
-
    -).
- wallet
- , for customizing the (optional) embedded wallet modal.* visible
-
    - , a Boolean determining whether or not the embedded wallet modal is shown or not. If set tofalse
-
    - , the user will rely on the functions you provide them within your application to interact with the generated wallet.
-
    - customStyle
-
    - , for more targeted, narrowly-defined customizations (ignore this if you're settingvisible
-
    - tofalse
-
    -).* supportChains
-
    -
        - , the chains you'd like the modal to support. In this case, we'll be usingArbitrumSepolia
-
    -
        - , imported from@particle-network/chains
-
    -
        - .

With these configurations set, Particle Auth Core will be initialized and, therefore, ready to be utilized within yourApp.tsx file. At this point, yourindex.tsx file (or its equivalent) should look similar to the example below.

import React from

'react' ; import ReactDOM from

'react-dom/client' ; import

{ ArbitrumSepolia }

from

'@particle-network/chains' ; import

{ AuthCoreContextProvider }

from

'@particle-network/auth-core-modal' ; import App from

'./App' ;

// Optional, not always needed import ('buffer') . then (({ Buffer })

=>

{ window . Buffer = Buffer ; }) ;

ReactDOM . createRoot (document . getElementById ('root')

as HTMLElement) . render (< React . StrictMode

< AuthCoreContextProvider options = { { projectId : process . env . REACT_APP_PROJECT_ID ,

// -- clientKey : process . env . REACT_APP_CLIENT_KEY ,

// Retrieved from https://dashboard.particle.network appId : process . env . REACT_APP_APP_ID ,

// -- erc4337 :

{ // Optional name :

'SIMPLE' ,

// The smart account you intend to use version :

'1.0.0' ,

// Leave as 1.0.0 unless you're using Biconomy V2 } , wallet :

{ // Optional visible :

true ,

// Whether or not the embedded wallet modal is shown customStyle :

{ supportChains :

[ArbitrumSepolia] ,

// Locking the modal to Arbitrum Sepolia } , } , } }

< App /

// Where you're utilizing Particle Auth Core < / AuthCoreContextProvider

< / React . StrictMode

,) ;

Building the application

Now that you've set up your project, installed the relevant dependencies, and configured Particle Auth Core, you're ready to move over to yourApp.tsx file.

WithinApp.tsx , you'll be defining all of the login methods for the application itself (such as the initiation of social login). As mentioned, this will be done by combining@particle-network/auth-core-modal and@particle-network/aa .

Managing hooks

@particle-network/auth-core-modal operates off of hooks such asuseEthereum (used to pull an EIP-1193 provider in this example),useConnect (to facilitate social login),useAuthCore (here used for retrieving user information post-login), etc.

Therefore, within the first few lines of yourApp component, you'll need to define the following objects derived from the hooks mentioned above:

- provider
- fromuseEthereum
- .
- connect
- anddisconnect
- fromuseConnect
- .
- userInfo
- fromuseAuthCore
- .

DefiningSmartAccount

While we could use these hooks on their own to facilitate interaction with the associated EOA directly, we'll be using a smart account here. Therefore, we'll need to create aSmartAccount object (imported from@particle-network/aa) using ourprovider (which is attached to our EOA) object to define a Signer.

Outside ofprovider , we'll pass the following parameters into our new instance ofSmartAccount :

- projectId
- ,clientKey

- , andappId
- . These will be the same values you used withinAuthCoreContextProvider
- , retrieved from theParticle dashboard
- .
- aaOptions
- , used to configure the smart account implementation you'd like to use. This containsaccountContracts
- , which takes:* SIMPLE
  - (orBICONOMY
  - ,LIGHT
  - ,CYBERCONNECT
  -), containing:* chainIds
    - , the chains you plan on using. In this case, we'll useArbitrumSepolia.id
    - .
    - version
    - , the version of the smart account you're using. Similar toAuthCoreContextProvider
    - , this should be1.0.0
    - unless you use Biconomy V2.

Therefore, your instance ofSmartAccount should follow the structure within the example below.

const smartAccount =

new

SmartAccount (provider ,

{ projectId : process . env . REACT_APP_PROJECT_ID , clientKey : process . env . REACT_APP_CLIENT_KEY , appId : process . env . REACT_APP_APP_ID , aaOptions :

{ accountContracts :

{ SIMPLE :

[{ chainIds :

[ArbitrumSepolia . id] , version :

'1.0.0'

}] , } , } , }) ; The object you save this instance within (smartAccount in the snippet above) can be used alone to construct and execute UserOperations; although, in this example, we'll be plugging it into a custom Ethers object. By doing this, we'll be able to construct and send transactions as within any other standard application scenario, although instead routing them through an MPC-powered smart account.

This will be done throughAAWrapProvider from@particle-network/aa , allowing us to convertsmartAccount into an EIP-1193 provider object to be used withinethers.providers.Web3Provider , as shown below.

const customProvider =

new

ethers . providers . Web3Provider (new

AAWrapProvider (smartAccount , SendTransactionMode . Gasless) , 'any' ,) ; Within the constructor ofAAWrapProvider , we're also usingSendTransactionMode.Gasless (which can be imported from@particle-network/aa) to ensure that all

UserOperations sent through this object (customProvider) will be sponsored (or attempted to be).

Initiating social login

Using Particle Auth Core, facilitating social logins is quite simple. Calling back to theconnect function defined fromuseConnect earlier, we'll be wrapping this within a simple function,handleLogin .

connect will work on its own, handling the entire social login process; although to prevent it from being called while a user may already be logged in, we'll place it behind a conditional withinhandleLogin .

This conditional will check the truthy/falsy value ofuserInfo (fromuseAuthCore), which will remain undefined until a user has successfully logged in. Thus, we can callconnect on the condition thatuserInfo is undefined (indicating that they have yet to log in).

Withinconnect , we'll need to define two key parameters:

- socialType
- , the social login mechanism you'd like to use (such as 'google', 'twitter', 'email', 'phone', 'github'). If this is left as an empty string, a generalized authentication modal aggregating every option will be shown.
- chain
- , the chain we'll be connecting to. This should beArbitrumSepolia
- (orArbitrumOne
- ,ArbitrumNova
- ,ArbitrumGoerli
-).

Upon callinghandleLogin , the user will be taken through the defined social login mechanism, after which an EOA will be generated (using MPC-TSS) and assigned to the smart account configured previously.

Your usage ofconnect may look something like this (although, as mentioned, it can be used in isolation if preferred).

const

handleLogin

=

async

(authType)

=>

{ if

(! userInfo)

{ await

connect ({ socialType : authType , chain : ArbitrumSepolia , }) ; } } ;

Executing a gasless transaction

At this point, a user has logged in through their social account and been assigned a smart account. We'll need to build a simple function to test it, specifically through a gasless transaction.

Because we're using Ethers, this can be a simple transaction construction and execution, using thesendTransaction method on asigner object from{your provider object}.getSigner() .

This method will take a standard transaction object. In this case, we'll simply define an object containing:

- to
- , the recipient of the transaction. We can set this to0x0000000000000000000000000000000000000dEaD
- for the sake of the demo.
- value
- , the amount of ETH you'd like to send to the recipient address.

This object (we'll define it astx) can then be passed intosigner.sendTransaction(tx) . Upon calling, the user will be asked to confirm a transaction (sign a UserOperation hash) through an embedded popup within your application. After doing so, the transaction will be executed on-chain.

Following the example described above, your function may look similar to the snippet shown below.

const

executeUserOp

=

async

( )

=>

{ const signer = customProvider . getSigner ( ) ;

const tx =

{ to :

'0x000000000000000000000000000000000000dEaD' , value : ethers . utils . parseEther ('0.001') , } ;

const txResponse =

await signer . sendTransaction (tx) ; const txReceipt =

await txResponse . wait ( ) ;

notification . success ({ message :

'Transaction Successful' , description :

(< div

Transaction Hash : { ' ' } < a href = {https://sepolia.arbiscan.io/tx/ { txReceipt . transactionHash } } target = "_blank" rel = "noopener noreferrer"

{ txReceipt . transactionHash } < / a

< / div

) , }) ; } ; This transaction will be gasless because we're meeting two conditions. The first is whether or not we've expressed that this transaction should be sponsored (done earlier by includingSendTransactionMode.Gasless withinAAWrapProvider). The second condition is the existence of adequate funds to sponsor the transaction. Because we're on a Testnet (Arbitrum Sepolia), all transactions are automatically sponsored without the need to deposit USDT. However, if this were on Arbitrum One or Arbitrum Nova, the Paymaster shown on theParticle dashboard would need to be funded.

Conclusion

Building an application on Arbitrum that takes advantage of social logins and smart accounts simultaneously only takes a few lines of code, and is even more succinct if you're already using Ethers, Web3.js, or any other standard library that supports EIP-1193 providers.

To view the complete demo application leveraging the code snippets covered throughout this document, take a look at the GitHub repositoryhere .

Additionally, to learn more about Particle Network, explore the following resources:

- Website:https://particle.network
- Blog:https://blog.particle.network
- Documentation:https://developers.particle.network Edit this page Last updatedonMar 15, 2024 Previous PARSIQ Next How to use Supra's price feed oracle