Tutorial: How to integrate Astroport swaps into your web-app

October 20, 2022 ✦ Technical

In previous tutorials, we covered various Astroport contracts that make up the core of the protocol. However, oftentimes, developers are building user-facing applications that require further integrations with existing contracts. For example, building a new UI for Astroport, integrating Astroport swaps in web wallets, building a DEX aggregator on top of Astroport, and so on. In the first tutorial of this new series, we will walk through a general strategy for implementing Astroport swaps directly into your web app. This will tie previous concepts from our smart contract series and push the code closer to production.

Strategy

When it comes to swapping tokens, there are several layers that are at work in the background, such as wallet functionality, fee calculations, and more. Although contracts remain decentralized and platform agnostic, oftentimes, web applications become centered around a particular wallet and its message types to broadcast and transact. This tutorial does not focus on any specific wallet or implementation and instead takes a more abstract approach allowing developers to have flexibility in their integration. For example, you might be creating your own wallet directly and just need to understand how the swap integration works. In future tutorials in this series, we will discuss wallets, estimating fees, and other topics in depth.

Before we swap tokens, we will also need to simulate our swap. This will return information that is required as inputs in our actual swap message (namely max spread and belief price).

The recommended strategy is to use a modular architecture like React components, with the simulation and swap functionality as a hook. Your index.ts file will handle user-facing logic. However, this strategy is flexible enough to be implemented using another framework.

Let's begin.

useSwap

In this tutorial, we will be using a useSwap hook that stores both our simulation query and our swap executable message. First, our simulation query. At the most abstract layer, we need a function to call our client using smart contract queries. This function will take in a pool address and the query message itself. In this case, the simulation message consists of an offer asset (amount and contract address).

```
const response = await client.queryContractSmart(
      poolAddress,
      {
        simulation: {
          offer_asset: {
            amount: String(amount *
TOKEN_DECIMALS),
            info: { token: { contract_addr:
offerAssetAddress } },
          },
        },
      }
  );
```

Note that the amount is multiplied by a constant (TOKEN_DECIMALS) to normalize the value. In the case of ASTRO/xASTRO, the tokens have 6 decimal places.

const TOKEN_DECIMALS = 1000000;

In a React hook, this logic could look something more like this:

```
function useSwapSimulate({ amount,
offerAssetAddress, poolAddress, wallet }:
SimulateProps) {
 return useQuery(['swap-simulate', amount,
offerAssetAddress, poolAddress], async () => {
   if (!amount || amount <= 0 ||
!offerAssetAddress || !poolAddress || !wallet) {
     return null;
   }

   const client = await
CosmWasmClient.connect(wallet?.network.rpc ||
"");

   const response = await
client.queryContractSmart(
     poolAddress,
     {
       simulation: {
         offer_asset: {
           amount: String(amount *
TOKEN_DECIMALS),
           info: { token: { contract_addr:
offerAssetAddress } },
         },
       },
     }
   );

   return {
     amount: Number(response.return_amount) /
TOKEN_DECIMALS,
     commission:
Number(response.commission_amount) /
TOKEN_DECIMALS,
     spread: Number(response.spread_amount) /
TOKEN_DECIMALS,
     price: (amount * TOKEN_DECIMALS /
Number(response.return_amount)),
   };
 }, {
   enabled: !!amount && amount > 0 &&
!!offerAssetAddress && !!poolAddress && !!wallet,
 });
}
```

Let's break this down:

Our function takes in several SimulateProps which we have defined as:

```
type SimulateProps = {
    amount: number;
    offerAssetAddress: string;
    poolAddress: string;
    wallet: WalletConnection | null;
};
```

Depending on your wallet, your implementation may look a bit different, but all calls to the simulation query will need more or less the same type of information. In our example, here is where we connect to our client:

```
const client = await
CosmWasmClient.connect(wallet?.network.rpc ||
"");
```

Lastly, note that we are also specifying the type of information we would like returned. This will be useful in performing our actual swap:

For the swap itself, the structure is quite similar. First the code:

```
export default function useSwap({ amount,
offerAssetAddress, poolAddress, slippage = 0.005,
wallet }: Props) {
 const simulate = useSwapSimulate({ amount,
offerAssetAddress, poolAddress, wallet });
   return [
   {
     type:
"/cosmwasm.wasm.v1.MsgExecuteContract",
     sender: wallet?.account.address || "",
     contract: offerAssetAddress,
     msg: {
       send: {
         amount: String(amount * TOKEN_DECIMALS),
         contract: poolAddress,
         msg: toBase64({
           swap: {
             max_spread: String(slippage),
             belief_price:
String(simulate.data?.price || 1),
           },
         }),
       },
     }
   }
 ];
}
```

Some notes:

- Unlike our simulation function, we are exporting our useSwap function to use later in our application.
- Our props are mostly the same. In this example, we set a default value for our slippage set to 0.005. This prevents our belief_price from deviating too much from the actual price when performing our swap.
- We call the simulate function within our useSwap function and use our props as inputs. Remember, this query returns useful information we can call later in our function.
- Every function, despite wallet differences, will have to specify the type of message, information about the sender, the contract that is being interacted with, and the message itself.
- The executable message contains an amount to swap, the Astroport pool address to swap tokens between, and a Base64 encoded message.
- We use the following function to encode our message:
- Lastly, our max_spread and belief_price are plugged into our function.

index.ts

Now that we are done with our useSwap component, we can move on to our index.ts file. In this section, we will work on calling our hook and integrating it with a front-end app.

First, we will define the contracts that we will be working with. For our purposes, we will limit it to ASTRO-xASTRO swaps:

```
type ASTROPORT_CONTRACTS = {
 [key: string]: {
   astroContractAddress: string;
   xAstroContractAddress: string;
   astroxAstroPoolAddress: string;
 };
};

const CONTRACTS: ASTROPORT_CONTRACTS = {
 "phoenix-1": {
   astroContractAddress:

"terra1nsuqsk6kh58ulczatwev87ttq2z6r3pusulg9r24mf
j2fvtzd4uq3exn26",
   xAstroContractAddress:

"terra1x62mjnme4y0rdnag3r8rfgjuutsqlkkyuh4ndgex0w
l3wue25uksau39q8",
   astroxAstroPoolAddress:

"terra1muhks8yr47lwe370wf65xg5dmyykrawqpkljfm39xh
kwhf4r7jps0gwl4l",
 },
 "pisco-1": {
   astroContractAddress:

"terra167dsqkh2alurx997wmycw9ydkyu54gyswe3ygmrs4l
wume3vmwks8ruqnv",
   xAstroContractAddress:

"terra1ctzthkc0nzseppqtqlwq9mjwy9gq8ht2534rtcj3yp
lerm06snmqfc5ucr",
   astroxAstroPoolAddress:

"terra19gn0pd6a7n8kgmdg8h76t70rzssf0dguvetjfkq5y7
8v8cr6fk3s7jfvgl",
 },
};
```

Next, we define a variable that calls our useSwap hook that we just created. We use our CONTRACTS constant as an input for our offerAssetAddress and poolAddress props:

```
const swap = useSwap({
 amount,
 offerAssetAddress:
   CONTRACTS[recentWallet?.network.chainId ||
"phoenix-1"]
     .astroContractAddress,
 poolAddress:
   CONTRACTS[recentWallet?.network.chainId ||
"phoenix-1"]
     .astroxAstroPoolAddress,
 wallet: recentWallet,
});
```

Now we broadcast our transaction! Our transaction takes in our executable message (swap.msgs from above), the feeAmount, and gasLimit (the last two will be discussed in a future article):

```
const onSwap = () => {
 broadcast({
   messages: swap.msgs,
   feeAmount: feeEstimate?.fee.amount,
   gasLimit: feeEstimate?.gasLimit,
 })
   .then((result) => {
     console.log("Broadcast result", result);
     setAmount(0);
     astroBalance.refetch();
     xAstroBalance.refetch();
   })
   .catch((error) => {
     console.error("Broadcast error", error);
   });
};
```

Note that our function also logs our results and refetches our balance after the transaction is complete.

Lastly, to use this onSwap function in our actual UI, we use a Swap button to broadcast our transaction (note that you will have to implement further logic outside of the scope of this article to sign and confirm the transaction):

```
<button
onClick={onSwap}
disabled={
 astroBalance.data <= 0 ||
 amount <= 0 ||
 amount > astroBalance.data
}
>
Swap
</button>
```

That's it! We simulated and performed the swap functionality as isolated hooks. We called our hook in our index.ts application logic. And we plugged in our application logic directly into a button.

Stay tuned for future tutorials that build upon this series!

✦ Follow Astroport on Twitter and subscribe to the Astroport email newsletter to get the latest alerts from the mothership.

DISCLAIMER

Remember, Terra 2.0 and Astroport are experimental technologies. This article does not constitute investment advice and is subject to and limited by the Astroport disclaimers, which you should review before interacting with the protocol.