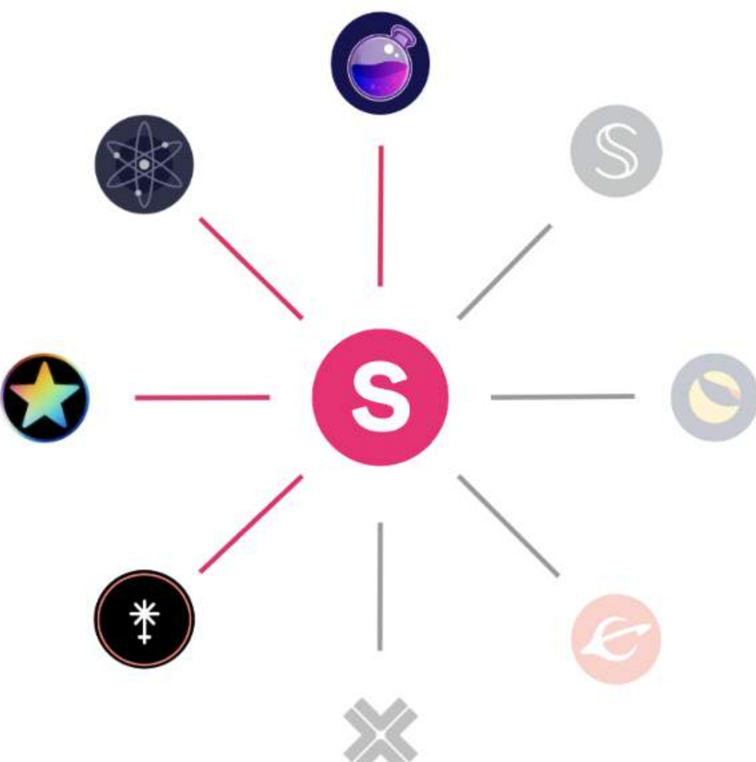
Twitter | Discord | Website

Stride is a blockchain ("zone") that provides liquidity for staked tokens. Using Stride, you can earn *both* staking *and* DeFi yields across the Cosmos IBC ecosystem. Read our "Meet Stride" blog post to learn more about why we built Stride.

Users can liquid stake their tokens on any Cosmos chain using Stride (a sovereign zone).

Currently, Stride supports liquid staking for:

- Cosmos Hub (**stATOM**)
- Osmosis (**stOSMO**)
- Juno (**stJUNO**)
- Stargaze (**stSTARS**)
- Evmos (**stEVMOS**)
- Terra 2 (**stLUNA**)
- Injective (**stINJ**)
- Umee (**stUMEE**)
- Comdex (**stCMDX**)
- IBCX (**stIBCX**)



Users receive staked tokens immediately when they liquid stake. Rewards accumulate in real time to staked token holders. These staked tokens can be freely traded, and can be redeemed with Stride at any time to receive back native tokens plus staking rewards.

On the backend, Stride permissionlessly stakes these tokens on the host chain and compounds user rewards.Users continue to earn staking yield, and can earn additional yield by lending, LPing, and more.

Users can always redeem from Stride. When they redeem, Stride initiates unbonding on the host zone. Once the unbonding period elapses, users receive native tokens in their wallets.

The process to onboard new chains is simple; anyone can propose onboarding a new chain through a governance vote, which will automatically onboard the new chain if passed.

Stride plans to rapidly expand reach throughout the Cosmos ecosystem. The chains and tokens that we plan to onboard in the next 12 months year are:

Secret (**stSCRT**), Kava (**stKAVA**), Oasis (**stROSE**), Axelar (**stAXL**), Akash (**stAKT**), Regen (**stREGEN**), Sommelier (**stSOMM**), Band (**stBAND**), dYdX (**stDYDX**), , Kujira (**stKUJI**), E-Money (**stNGM**), Crypto.Org (**stCRO**), Sifchain (**stEROWAN**), Crescent Network (**stCRE**), MediBloc (**stMED**), Persistence (**stXPRT**), Iris (**stIRIS**), AssetMantle (**stMNTL**), Sentinel (**stDVPN**), BitSong (**stBTSG**), Cheqd (**stCHEQ**), Chihuahua (**stHUAHUA**), KiChain (**stXKI**), Ixo (**stIXO**), Microtick (**stTICK**), Fetch.Ai (**stFET**), Konstellation (**stDARC**), Desmos (**stDSM**), Bitcanna (**stBCNA**), Lum Network (**stLUM**), Bostrom (**stBOOT**), Likecoin (**stLIKE**), Dig Chain (**stDIG**), RiZON (**stATOLO**), OmniFlix (**stFLIX**), Decentr (**stDEC**), Vidulum (**stVDL**), Altered Carbon (**stACB**), Shentu (**stCTK**), and others.

Many more tokens will launch in the cosmos ecosystem over the coming months and years. Stride plans to support all IBCv3-compatible tokens as "st"-Tokens.

Stride is built using Cosmos SDK and Tendermint and created with Ignite. Under the hood, Stride leverages:

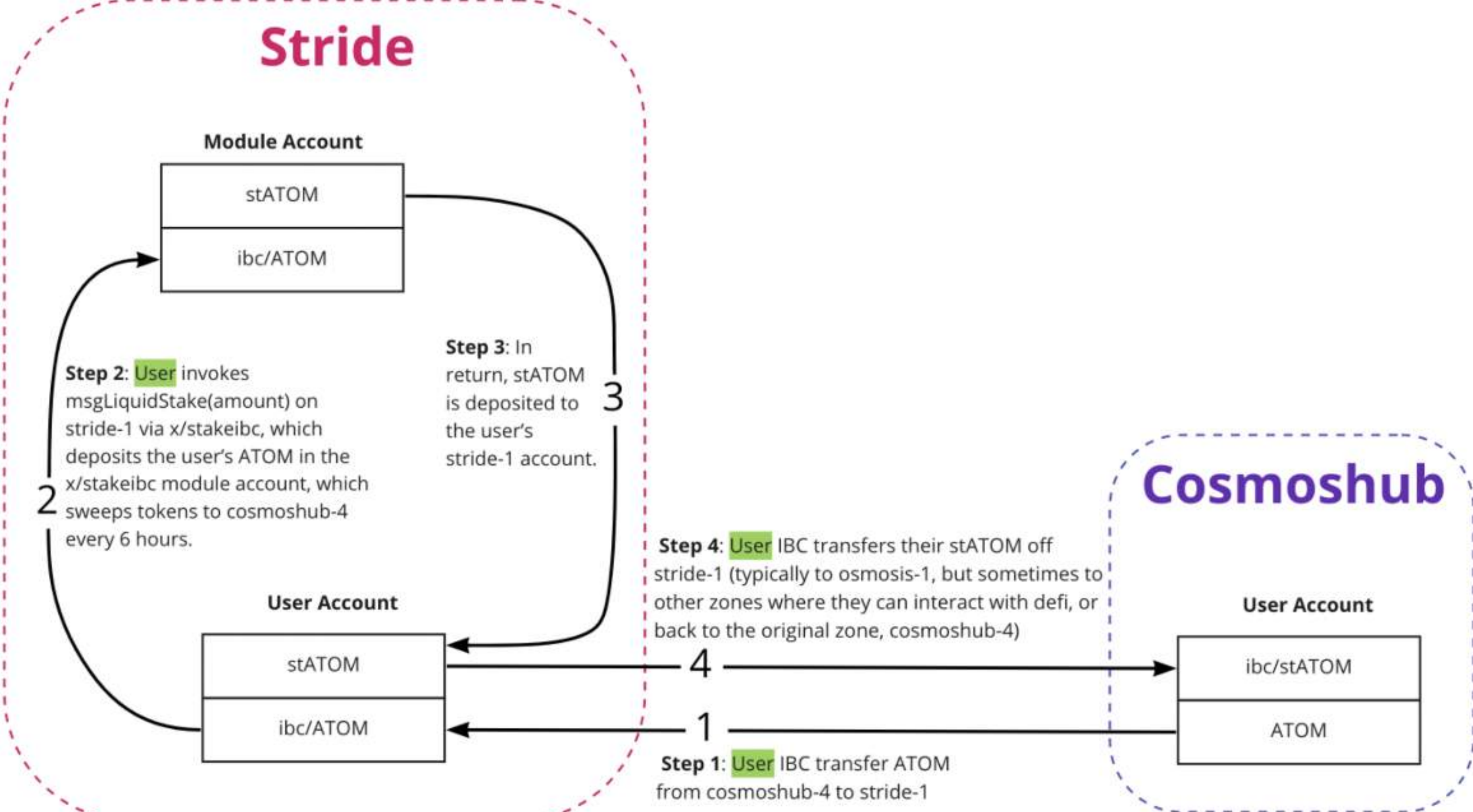
- Inter-Blockchain Communication protocol
- Interchain Accounts
- Interchain Queries

Stride's Technical Architecture

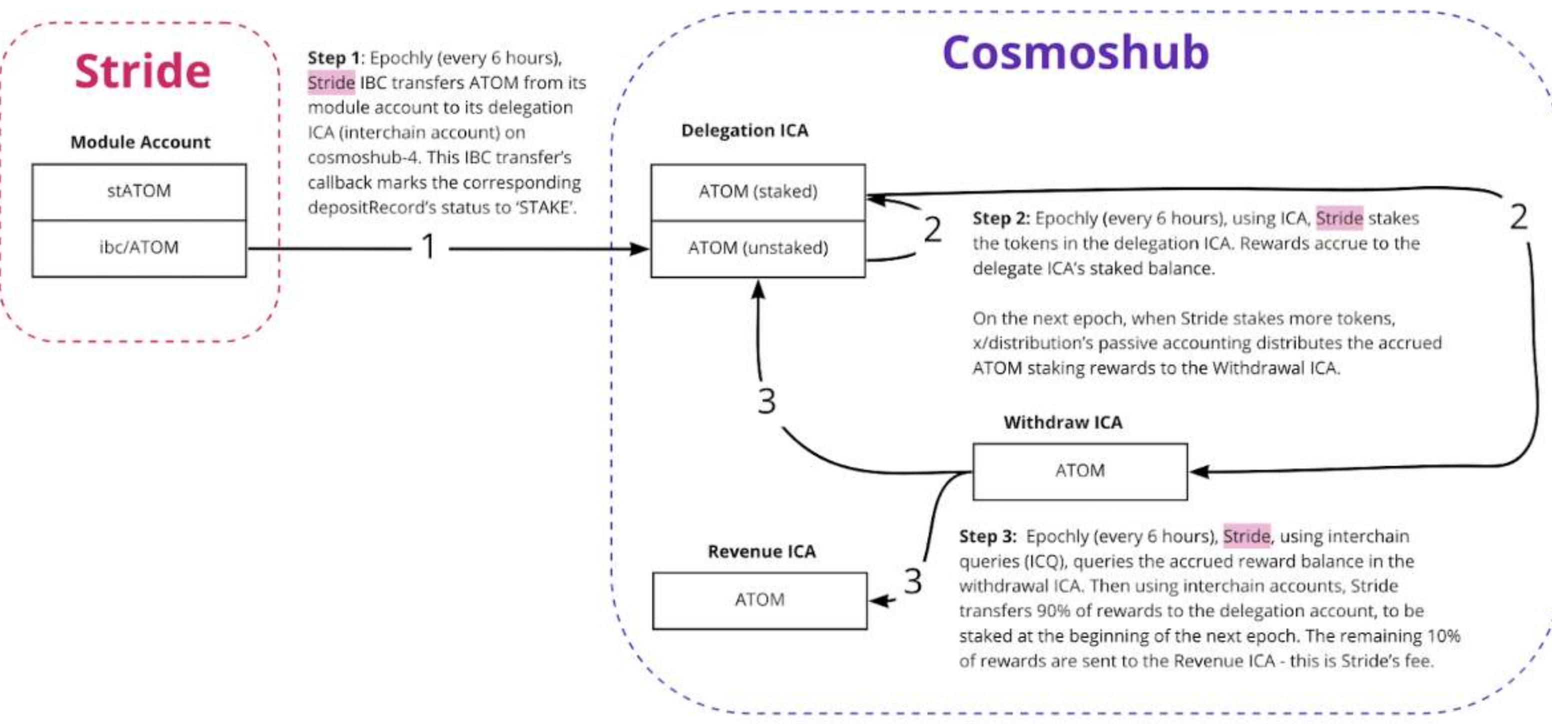
A technical overview of how Stride enabled interchain liquid staking.

High-Level System Design

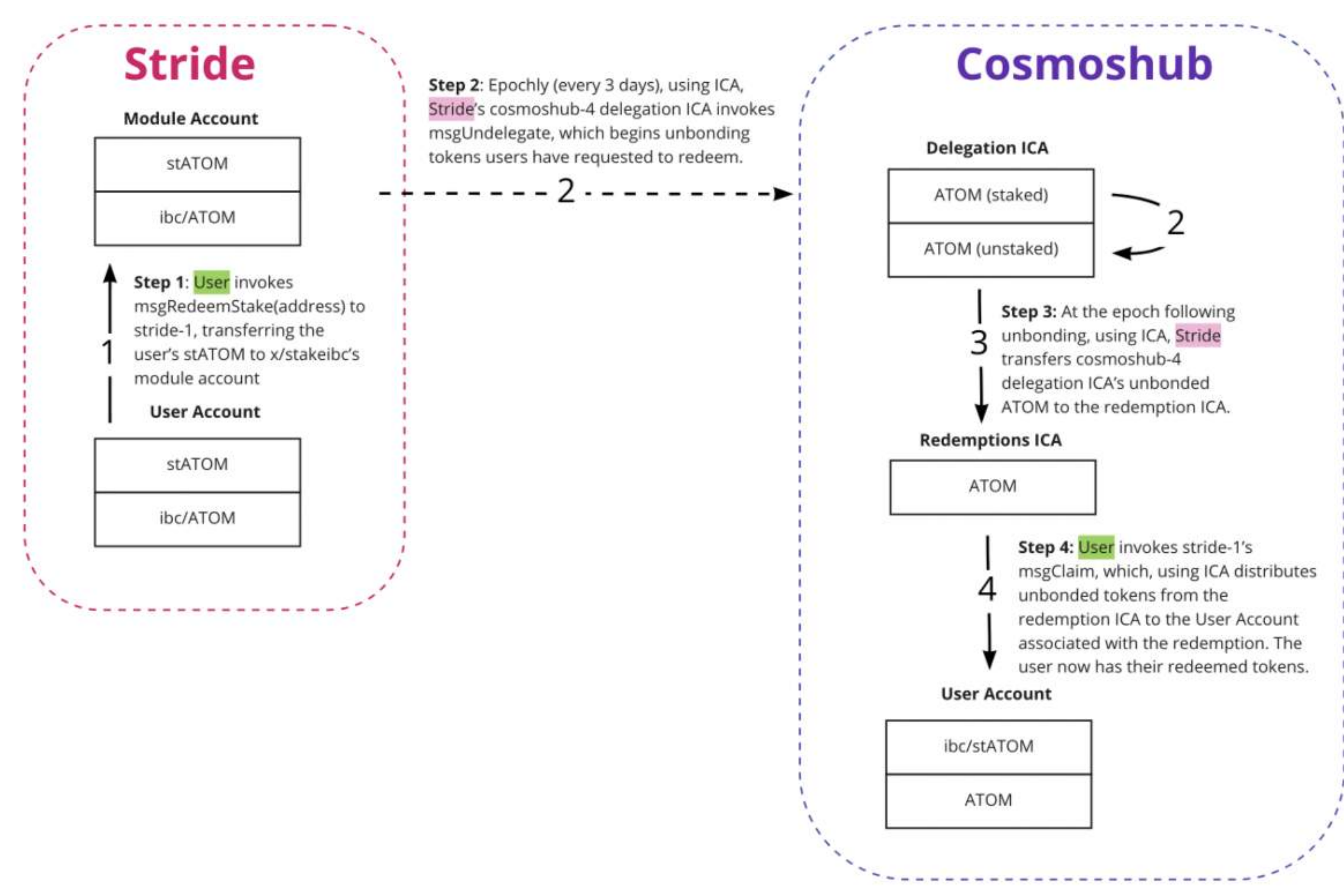
Users stake their tokens on Stride from any Cosmos chain. Rewards accumulate in real time. No minimum. They will receive staked tokens immediately when they liquid stake. These staked tokens can be freely traded, and can be redeemed with Stride at any time to receive your original tokens plus staking rewards.



On the backend, Stride permissionlessly stakes these tokens on the host chain and compounds user rewards. Stride lets users use your staked tokens to compound their yields. Continue to earn staking yield, and earn additional yield by lending, LPing, and more. They can set their own risk tolerance in Cosmos DeFi.



Users can always redeem from Stride. When they select "redeem" on the Stride website, Stride will initiate unbonding on the host zone. Once the unbonding period elapses, the users will receive native tokens in their wallets.



In-Depth System Design

**Written by the wonderful Informal Systems audit team.*

In Stride, staking occurs every 6 hours and it goes through 3 epochs:

- **epoch n**: New deposit record (DR) which tracks all deposits in a given epoch for a given host zone is created with 0 tokens. LiquidStake is called then and stTokens are minted, but the actual staking of the user's tokens is addressed in epoch n+2.
- **epoch n+1**: All tokens on DR from epoch n are IBC transferred from Stride's module account to Delegation ICA. Whenever a change to delegation happens all of the rewards are withdrawn (to Withdraw ICA). • epoch n+2: Tokens on the DR are staked (by weight) across all (30 at the moment) Stride validators.

Reinvestment executes automatically and the rewards are auto-compounded on every epoch (6h). 90% of the rewards are being sent to Delegation ICA (reinvestment) and 10% to the Fee ICA (the only place where Stride charges the fees)

- **epoch n**: Queries Interchain Query (ICQ) to check balances of Withdraw ICA and creates a new record for those tokens.
- **epoch n+1**: Transfers tokens to Delegation ICA from the Withdraw ICA.
- **epoch n+2**: Stakes the tokens.

Unstaking executes every day. Only 7 concurrent unbondings are allowed (a constraint on Cosmos) on host zones for a delegator and validator pair.

- **epoch n**: EpochUnbondingRecord is created. It stores many HostZoneUnbondings (HZU), with one HZU per host zone (e.g. for CosmosHub we have 1 HZU per epoch). When the user sends 1stAtom to Stride (Stride now custodies this 1 stAtom) and specifies an address on the Cosmos Hub that the tokens should be sent, HZU is updated and UserRedemptionRecord (URR) is created (claim on user's tokens that they can trigger later once the tokens have unbonded).
- **epoch n+m** (m = unbondingPeriodOnHostZone / 7 + 1): For CosmosHub it happens every 4 days (unbPeriod = 21 days). MsgUndelegate is triggered and all of the pulled unbonding tokens are undelegated (MsgUndelegate ICAs are triggered across the validators Stride has delegated to). HZUs are updated with unbonding time.
- **epoch n+unbonding time**: Tokens are transferred to Redemption ICA account. The URR is updated so that the tokens are claimable and anyone can transfer tokens to the already specified address which is stored on URR. So this is an ICA call that transfers tokens back to the end user's account.

Deposit & Liquid Staking

Withdrawal and deposit belong to regular bank transfers (outside of Stride). After transferring native tokens to Stride, liquid staking can be processed and it includes:

1. Sending IBC/Tokens to stakeibc account
2. Minting stTokens to stakeibc account
3. Sending stTokens from stakeibc account to user account on Stride

Staking & Reinvesment

Staking and reinvestment steps:

1. Sweeps the deposit record (DR) marked `TRANSFER_QUEUE` from previous epochs. Under the hub, it con structs IBC `MsgTransfer` with 30min timeout. `TransferCallback` is also created which is been called `OnAcknowledgementPacket` or `OnTimeoutPacket`. In the case of nill ack or ack_error DR's status is set back to `TRANSFER_QUEUE` otherwise it becomes a candidate for delegation with `DELEGATION_QUEUE` flag.
2. Delegates DRs with status `DELEGATION_QUEUE`. It creates a set of `MsgDelegate` msgs (delegation to every validator from that host zone whose relative amount is positive). Each validator gets `targetAmount=valWeight*depRecordAmount / totalValWeight`. Also, `DelegateCallback` is defined. In the case of the happy ibc path, the zone's staked balance is increased by a delegated amount to each validator whose `delegationAmt` is updated and finally DR is removed.
3. Rewards are automatically sent to the Withdrawal ICA.
4. `WithdrawalBalanceCallback` executes ICA `SendTx` with `MsgSend` to Delegation ICA (90% reward) and Fee ICA (10% reward). It also adds `ReinvestCallback` which triggers as previous ones, and in the happy path, it creates a new DR with `DELEGATION_QUEUE` status (using `WITHDRAWAL_ICA` source this time, not STRIDE source) while the sad path is ignored.

Unbonding

Unbonding goes as follows:

1. User sends `RedeemStake` msg which creates `UserRedemptionRecord` (URR), calculates the amount of native tokens to get back: `nativeAmount=stAmount*redemptionRate`, updates HostZoneUnbonding (HZU) record and sends numStAtoms to module's account where they will eventually be burned after unbonding.
2. On every epoch (a day) initiates unbondings for all host zones with ICA `SendTx` containing `MsgUndelegate` (from Delegation ICA to all validators) and sets HZU status to `UNBONDING_IN_PROGRESS` (note: HZU is created at `RegisterHostZone` with initial status set to `UNBONDING_QUEUE`). If ICA SendTx fails, `UndelegateCallback` will roll back the HZU status to `UNBONDING_QUEUE`, otherwise it burns escrowed stTokens, updates: HZU status to `EXIT_TRANSFER_QUEUE`, HostZone staked balance and validator amounts.
3. The funds are in escrow for the unbonding period and then get automatically transferred to back to the Delegation ICA.
4. Once per epoch if the unbonding period has elapsed, all HZU's tokens with `EXIT_TRANSFER_QUEUE` are swept as a batch to Redemption ICA. If everything goes well, HZU status is set to `CLAIMABLE`, otherwise it's reset to `EXIT_TRANSFER_QUEUE` during `RedemptionCallback` execution.
5. Users can claim their unbonded tokes via `ClaimUndelegatedToken` which also sets claim pending to TRUE (to avoid double claims). ICA `SendTx` is constructed with `MsgSend` inside (from Redemption ICA to user address on host) with 10min timeout. If no ack errors, URR is removed and HZU's native token amount is decremented by the claimed value. Else, if timeot or ack failure, claim pending is set to false.

Interchain Accounts

It can be seen that many paths lead to ICA's `SendTx`, including both staking and unstaking, while all paths (except direct Msgs) have the same root - the `BeforeEpochStart` function.

There are only two scenarios for ICA's `RegisterInterchainAccount` to be called. The first one is when registering the host zone, 4 ICAs are created. The second one, in the case of timeout, if a channel closes, the controller chain must be able to regain access to registered interchain accounts by simply opening a new channel which is done through `RestoreInterchainAccount`. `IBCTransfer` is called at one place only from record's module when sweeping existing deposits from Stride to the Host Zones.

Getting Involved

You may be wondering how you can stay connected, get project updates, or explore opportunities to contribute. The solution is simple: join our community and participate in governance!

Stride continues to grow and develop into a stronger project because of the dedication and engagement of our vibrant community. Our community members have provided the foundation we needed to gain momentum in the Cosmos ecosystem. Our community has supported us by running validators during the testnet, offering feedback and suggestions for the project throughout its evolution, supporting users in the online servers, creating tutorials and translations, participating in governance, spreading the word about us, and so much more. To connect directly with other Stride users and receive technical support and/or participate in liquid staking-related discussion, join 22,000+ others in our Discord community!

To get the most out of what Stride has to offer you, check out our other online community spaces and stay up to date on the latest news from the team, upcoming integrations, governance proposals, and community events. Check out the links below:

Join the Stride Community

Main Twitter – Community Twitter – Discord – Official Telegram – Reddit

Validator Discussions

Join Discord → Request the "Validator" role → Join the validator channels!

Governance Discussions

Commonwealth - Discussion Threads, Polls, + Proposals

Join Discord → get "Verified" → View the #governance channel!

All Users: Start Here

Welcome to Stride. To help you get the most out of your Stride experience, we have provided this documentation to support the development of your understanding of and success using the Stride protocol to liquid stake your tokens and launch you into your DeFi activity.

Our community has grown exponentially to include many different types of Cosmos users. **To help guide your process, we have identified and designed a suggested User Flow for three broad categories of users:**

New Cosmonauts
This is you if you are new to the Cosmos ecosystem or to the concepts of DeFi, (liquid) staking, etc. You may have not heard of liquid staking tokens nor know how, so we have provided a pool of educational resources on Cosmos, DeFi, and (Liquid) Staking. Check out this **"Getting Started"** section to read about key concepts you'll want to understand before diving into Cosmos Defi. You'll also want to check out our "Community" section and join our community on social media!

Active Cosmonauts
This is you if you are active in the Cosmos ecosystem or run a validator in the network, have explored other projects and/or are familiar with liquid staking, DeFi use cases, etc. You may or may not have liquid staked using Stride before. In our **"Tutorials"** section, we provide guidance on the Stride-specific liquid staking process, as well as an in-depth overview of relevant topics including unstaking/unbonding and adding liquidity on DEXs.

Advanced Users
This is you if you are a long-time Cosmos enthusiast, a developer or validator, have direct involvement in other projects, and/or want to contribute ideas to cutting edge discussions in the community. Take a look through the resources in the "Validators & Developers," and "Additional Topics" sections and plug into the conversation about high level topics driving the Cosmos ecosystem right now. Be vocal, contribute ideas, and help shape the future of Cosmos!

*If your project is interested in integrating with Stride, check out the documentation in our **"For Integration Partners"** section.*

What is Cosmos?

If you are a Cosmos or liquid staking beginner, begin your dive into the Cosmos using the resources below. This will set you up for success when you begin the liquid staking process.

Cosmos is a decentralized network of independent parallel blockchains that can scale and interoperate with one another. Cosmos is built on a Proof-of-Stake consensus mechanism. It uses a Proof-of-Stake consensus mechanism, in which validators rather than miners validate transactions and verify the accuracy of new blocks.

Cosmos is often referred to as the "Internet of Blockchains" because it enables blockchains to connect and communicate in a decentralized way using inter-blockchain technology (IBC). You can think of Cosmos as a huge network of interconnected tools that are modular, adaptable and communicate with each other as lego blocks to build a greater whole than their parts.

Blockchains built in the Cosmos ecosystem are called "zones". Cosmos zones do not necessarily need to connected to one another, but if they choose to, they have the ability to validate and keep record of each other using "light clients". Each zone can function on its own: it can authenticate accounts and transactions, create and distribute new tokens, execute changes by governance, and many other state transitions enabled by the Cosmos SDK modules.

Stride is a Cosmos zone. It is connected to the Cosmos Hub and many other Cosmos zones. Stride uses a new version of IBC ("IBCv3") to communicate, transact, and interoperate with other chains in the ecosystem. Stride is the first zone to use Interchain Accounts and Interchain Queries, two core IBC technologies.

Sources:
Cosmos Network
What is Cosmos?
About Cosmos
What is Cosmos? A beginner's guide to the Internet of Blockchains
Why Blockchains Need Cosmos Proof-of-Stake for a Sustainable Environment

What is (Liquid) Staking?

Stride is a liquid staking protocol - it allows you to use your staked tokens for other purposes, such as trading or providing liquidity, while still earning the staking yield. It's a bit like having the flexibility of a checking or brokerage account, while earning the higher yield of a savings account.

Proof-of-stake (PoS) blockchains require that tokenholders lock up their tokens in order to contribute to the PoS consensus mechanism, the method by which all transactions on the blockchain are verified. The greater the number of tokens staked on a network, the greater its security. Locking up tokens to provide this added security to the network is called staking. Stakers are compensated for their service through percentage-rate rewards on their staked tokens over time.

Staked tokens earn staking rewards but they are not liquid. While staked, they can't be used for other purposes, such as trading, lending or providing liquidity.

Most Cosmos zones offer users high levels of compensation for staking. Cosmos zone "staking yields" range from 20% (on the Cosmos Hub) to over 100% (on Evmos) per year. To earn these high yields, many users choose to stake their tokens: the percentage of total network tokens locked in staking tends to be quite high, sometimes more than 2/3 of all tokens.

Staking yields are so high that there is a meaningful opportunity cost to using your tokens in other applications (such as lending or providing liquidity), which often are less lucrative than staking. We believe that high staking yields and the resulting scarcity of liquid tokens in circulation has hamstrung the development of decentralized finance in Cosmos.

Enter liquid staking. Liquid staking makes staked tokens liquid, so that they remain staked, but can simultaneously be used in other applications. Liquid stakers enjoy both the staking rewards and the rewards offered by other applications, such as lending or providing liquidity.

In other words, liquid staking allows you to stake your tokens and earn staking rewards without having to give up control of your tokens. You can continue to use your tokens for other purposes, such as trading, lending or providing liquidity, while still earning the staking yield.

It's a bit like having the flexibility of a checking or brokerage account, while earning the higher yield of a savings account.

With liquid staking, decentralized finance applications built on blockchains with high staking yields no longer have to compete with attractive staking yields. We think that liquid staking is a prerequisite to a high-functioning, productive DeFi ecosystem in Cosmos.

Let's walk through an example. To liquid stake ATOM with Stride, you lock your ATOM on Stride Zone. Behind the scenes, Stride stakes this ATOM on your behalf. You receive stATOM in exchange. Unlike staked ATOM, stATOM is liquid, meaning it can be sold, transferred, or used in DeFi. stATOM is non-inflationary, so as rewards are compounding on your ATOM, your stATOM increases in value. At any time, you can redeem your stATOM for ATOM. You'll receive back more ATOM than you deposited, because you earned some staking rewards.

For a video overview of the liquid staking process with one of our founders, Vishal Talasani, check out this video.
Topics covered: what is Stride, liquid staking on Stride, unstaking, token value, integration with other platforms

Sources:
What is "proof of work" or "proof of stake"?
What is Staking?
Liquid Staking Coming to Cosmos

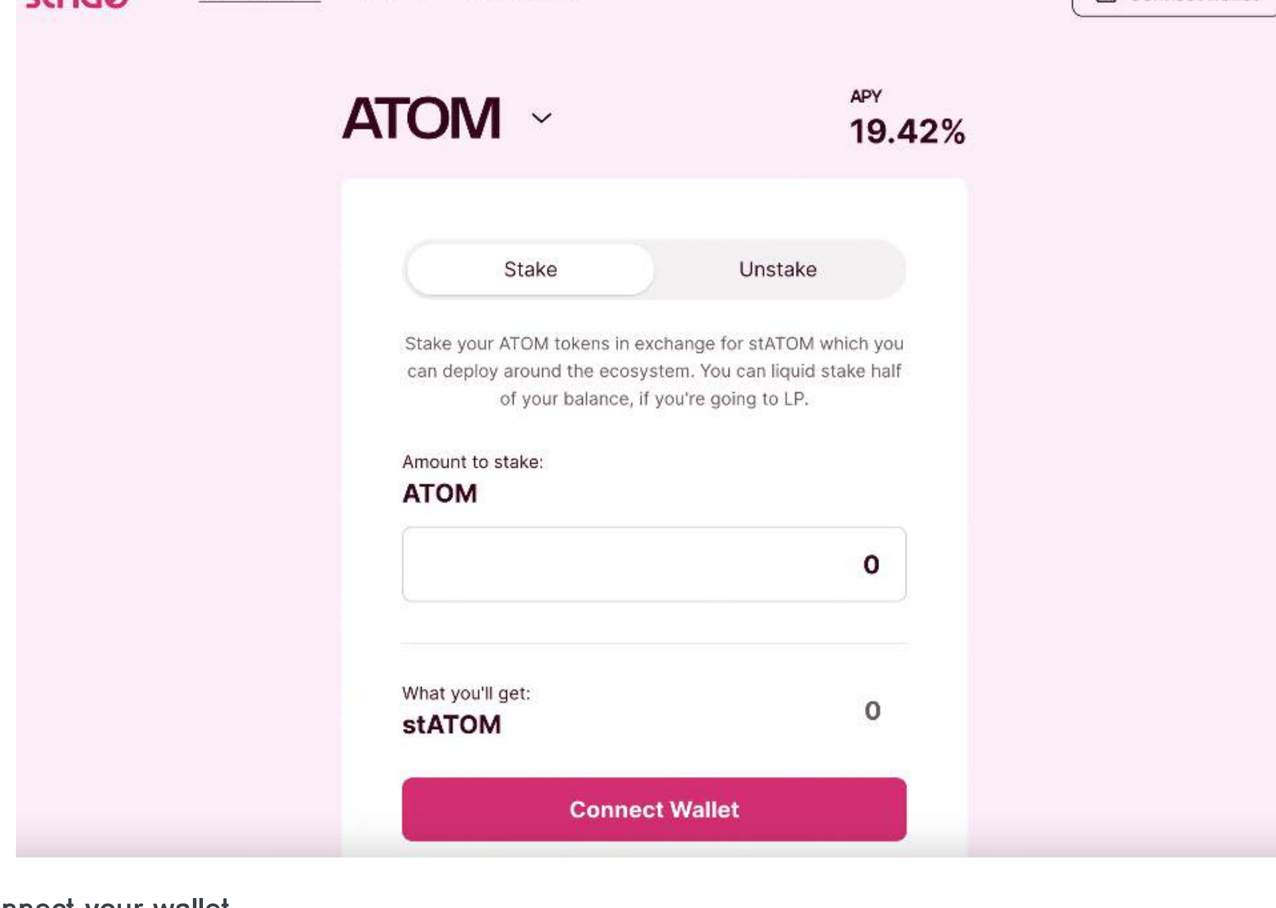
Liquid Staking on Stride

If you are already active in Cosmos and understand Staking on principle, you'll be successful using our app right away! Follow the guidance and relevant links below to get going.

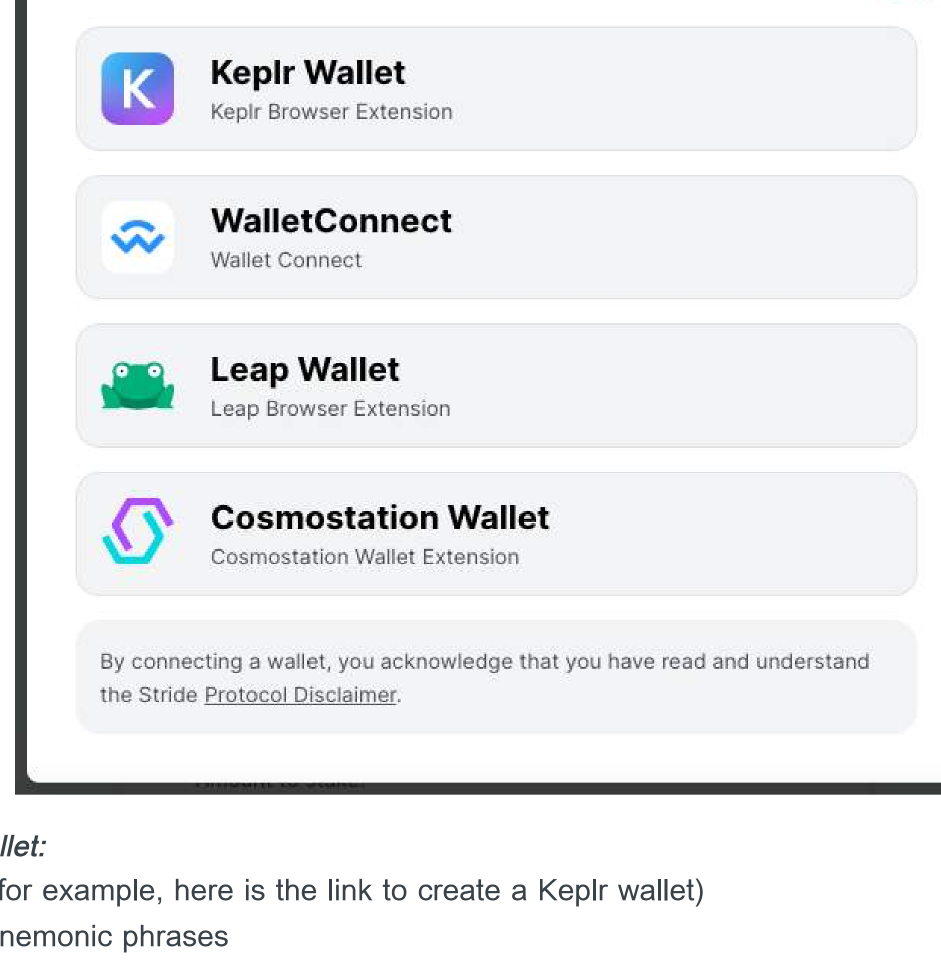
With Stride, you are now able to stake your tokens (ATOM, OSMO, JUNO, LUNA, EVMOS, STARS, INJ, UMEE, CMDX, IBCX) in exchange for stTOKENs which you can deploy around the Cosmos ecosystem. In order to liquid stake your tokens, visit stride.zone and select "Start liquid staking" or use the Stride App link https://app.stride.zone/.

To follow along with a video tutorial, **see this step-by-step screenshare tutorial here**. Otherwise, keep reading!

You will see this screen:



Step 1: Connect your wallet.
For this example we are staking ATOM tokens stored in a **Keplr wallet.** You will see the following options (select one):

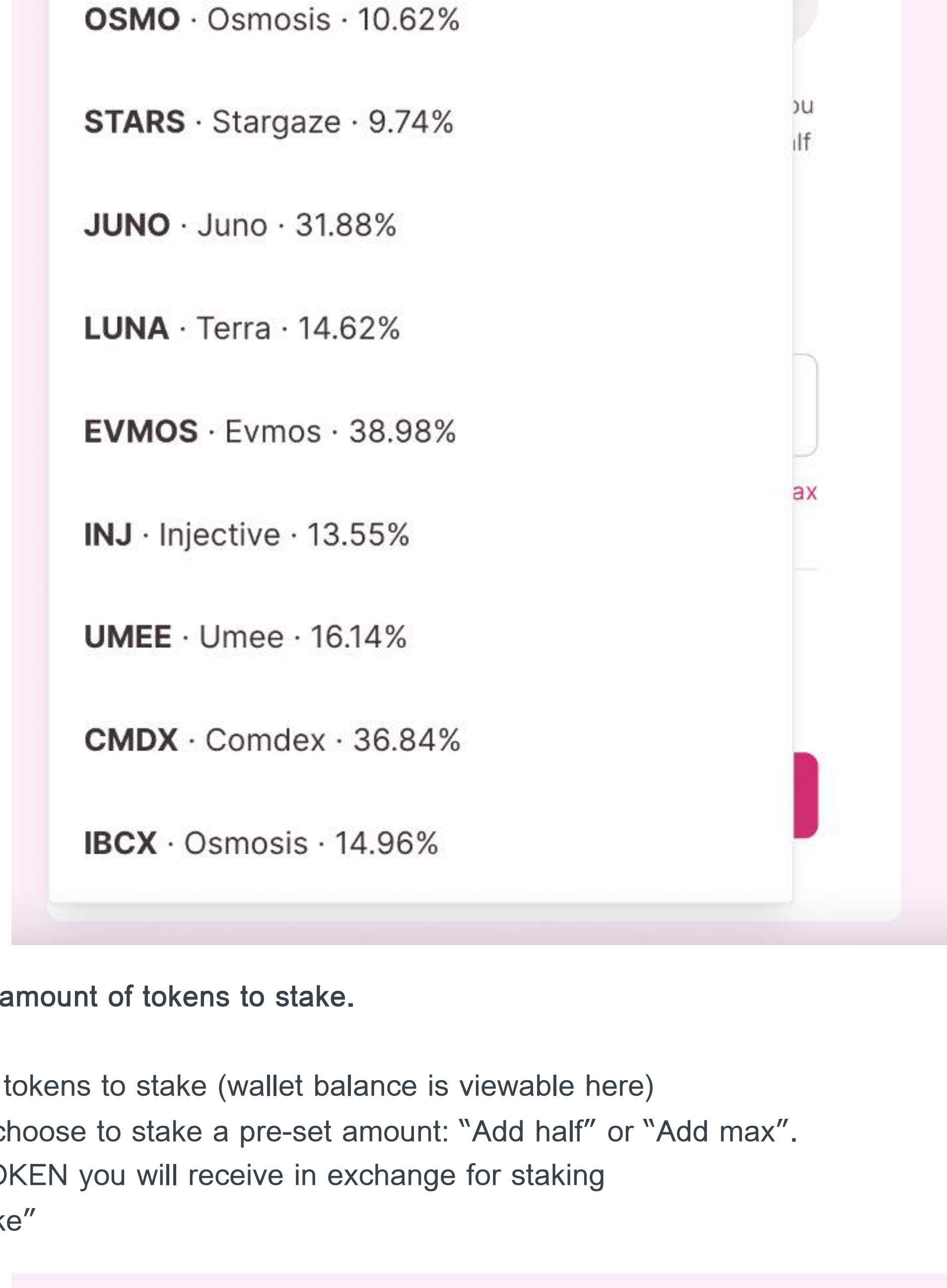


If you don't have a wallet:
→ create an account (for example, here is the link to create a Keplr wallet)
→ never share your mnemonic phrases
→ ensure sufficient supported token balance for staking (min. balance: 0.1)

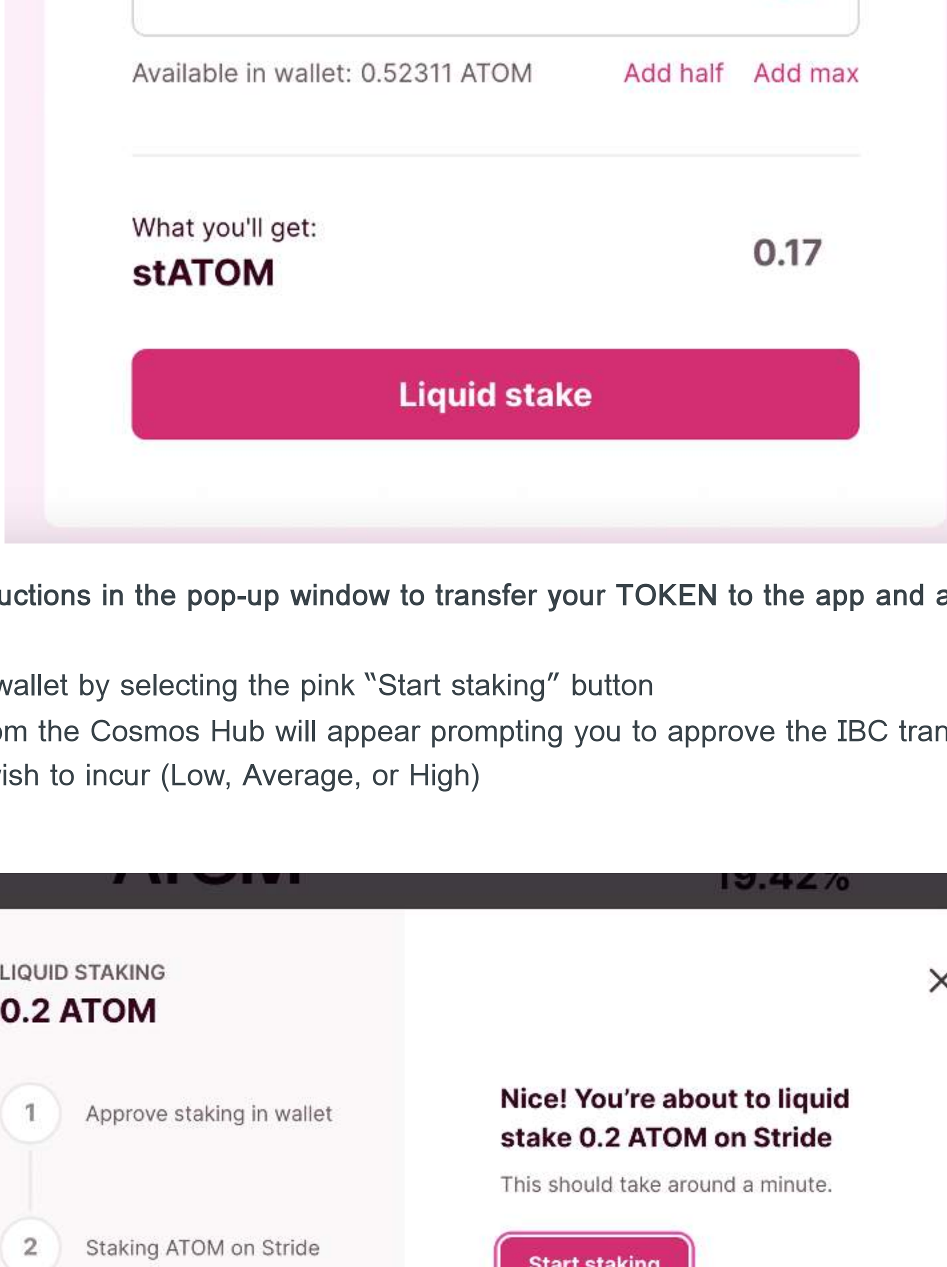
To connect your wallet on Stride:
→ select the pink "Connect Wallet" button in the top right corner
→ Enter your wallet password in a pop-up window
→ Once connected, you can see your wallet address in the top right corner of the screen

To change the token being staked:
→ Select the down arrow to the right of the bolded token (here we see ATOM)
→ choose a supported token from dropdown list

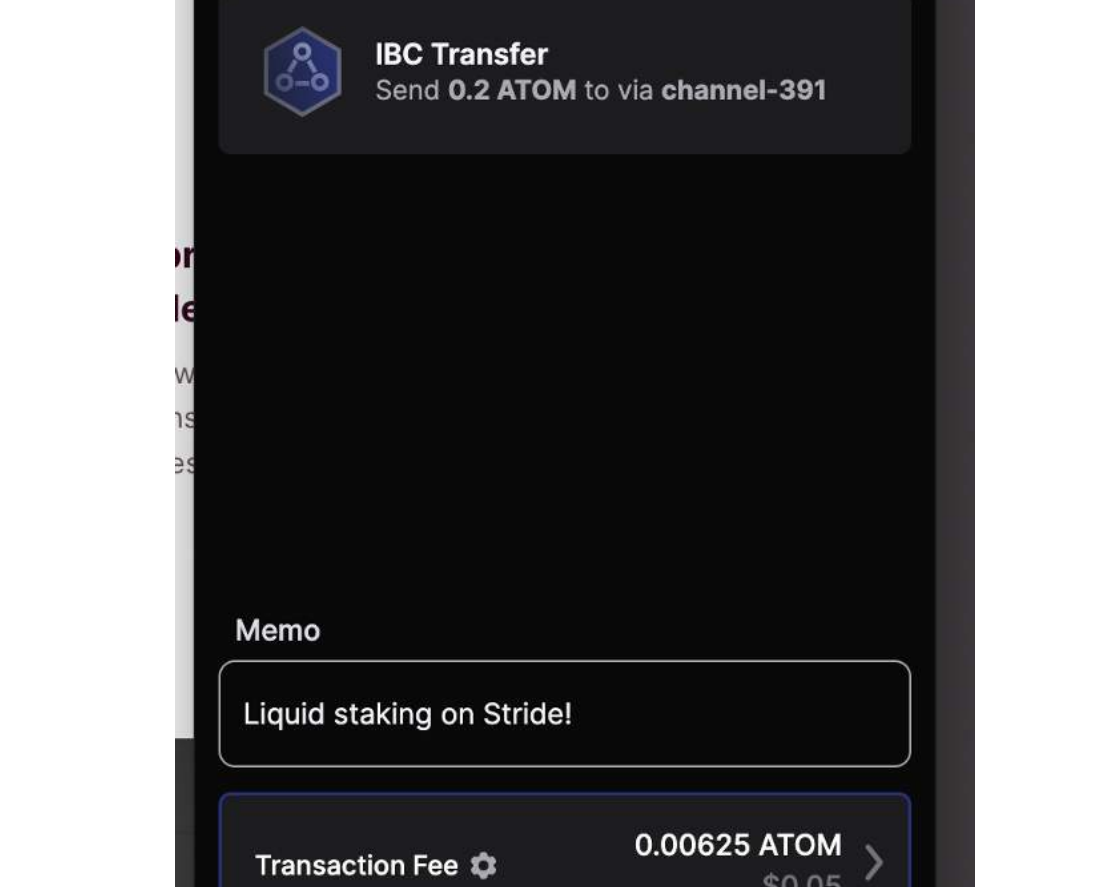
Supported tokens currently include:
ATOM, OSMO, JUNO, LUNA, EVMOS, STARS, INJ, UMEE, CMDX, and IBCX.



Step 2: Determine the amount of tokens to stake.
→ Click on "Stake"
→ input the amount of tokens to stake (wallet balance is viewable here)
→ manually input OR choose to stake a pre-set amount: "Add half" or "Add max".
→ See how much stTOKEN you will receive in exchange for staking
→ click on "Liquid Stake"

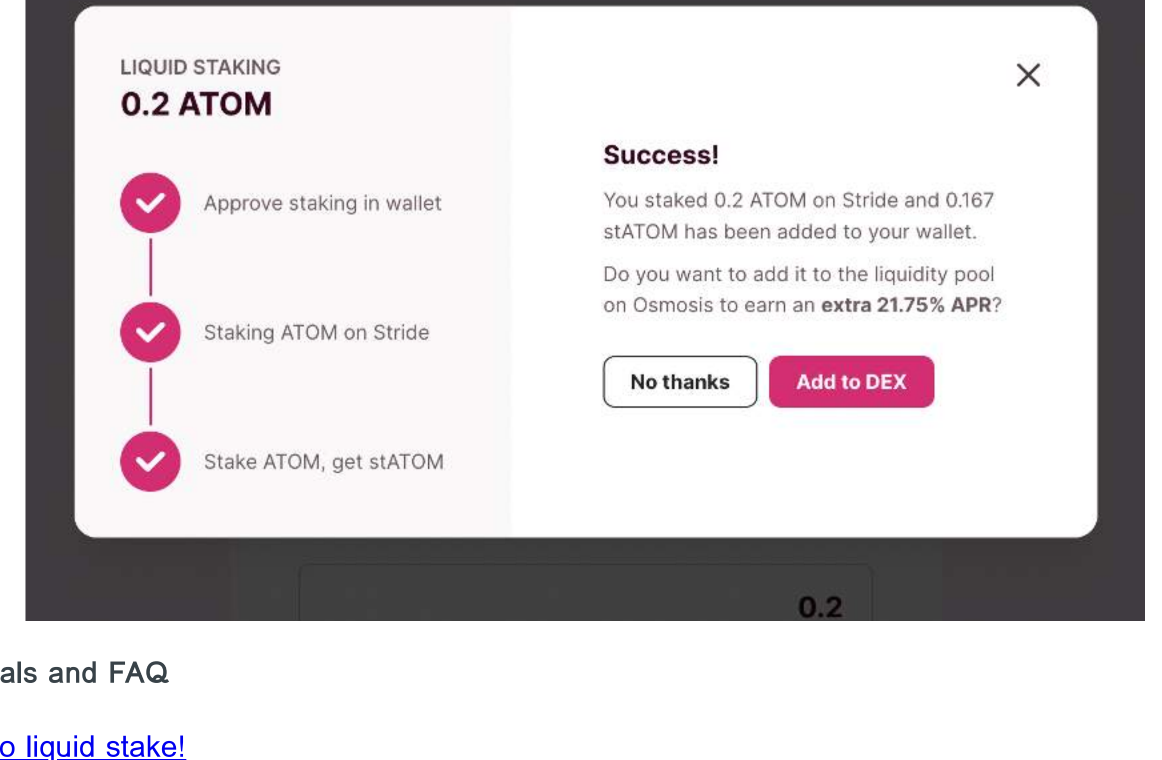


Step 3: Follow the instructions in the pop-up window to transfer your TOKEN to the app and approve the staking process.
→ Approve transfer in wallet by selecting the pink "Start staking" button
→ A pop-up window from the Cosmos Hub will appear prompting you to approve the IBC transfer
→ Select the fee you wish to incur (Low, Average, or High)
→ select "Approve"



After your token has been successfully transferred:
→ Approve staking in your wallet
You will see a different Cosmos Hub pop-up window
→ Select the fee you wish to incur (Low, Average, or High)
→ Select "Approve"
→ Review the 3 items on the staking checklist and ensure there is a pink checkmark next to each

You will be prompted to add your new stToken to a liquidity pool on one of our integration partner's DEXs. In this case, you are prompted to add your stToken to a liquidity pool on Osmosis. To earn extra APR, you can select "Add to DEX" and the transaction can happen via the Stride app interface. If you don't wish to do this, you can select "No thanks" and with this, complete the liquid staking process.



Complete Tutorials and FAQ

- Click here to liquid stake!
- Frequently Asked Questions
- Liquid Staking Guide - DonCryptonium
- Stride Blog Version: Liquid Staking Instructional Guide

If you need direct support troubleshooting or have a question that is not contained in our FAQ, please join our Discord server and open a Support Ticket.

Providing Liquidity on Osmosis

After learning how to liquid stake your tokens, learn how to **provide liquidity** in DeFi on DEXs such as Osmosis. We use Osmosis Pool #803 ATOM/stATOM for this example.

Adding Liquidity to a Pool on Osmosis

Now that you have staked your tokens, you can use your stTokens in decentralized finance. One way to do this is by providing liquidity to decentralized exchange (DEX) pools. For this tutorial, we will explain how to provide liquidity on Osmosis.

To see Osmosis' comprehensive guide to Providing Liquidity, click here.

For more background context and a Stride-specific tutorial, keep reading!

A Brief Overview of Liquidity Pools:
Liquidity is a measure of how easily an tokens can be exchanged to another tokens. A liquidity pool is a pool of digital tokens that allows trading on a decentralized exchange (DEX).

The primary goal of liquidity pools is to facilitate peer-to-peer (P2P) trading on DEXs. By providing a steady supply of buyers and sellers, liquidity pools ensure that trades can be executed quickly and efficiently.

Osmosis is the largest DEX in Cosmos and Stride's main liquidity hub.

Basically, Osmosis has two separate frontends.

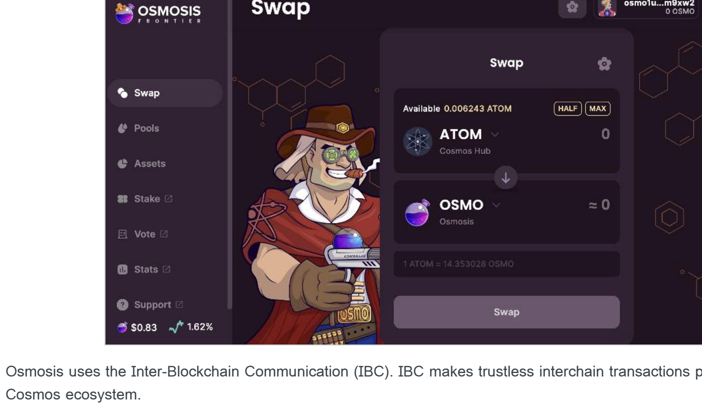
App Page:
- used for all on-boarded chains
- contains a lot of tokens that are registered onto the Osmosis AssetList Registry

Frontier:
- space for newly listed and unverified tokens
- multichain DEX currently used for Cosmos-based tokens
- users can access liquidity pools, swap tokens, generate and earn incentives

They look very similar - be sure that you are using the correct frontend depending on the type of token and pool you are interested in.

App Page:

Frontier:



Osmosis uses the Inter-Blockchain Communication (IBC). IBC makes trustless interchain transactions possible in the Cosmos ecosystem.

You can make a deposit into the platform directly from your wallet (we will use Keplr here) once it is connected. Once the funds are successfully deposited, you will be able to approve transactions and add liquidity to incentivized pools.

Explore the Stride DeFi integrations page

You can access this page from the Stride app page at the top. You will get to this link:

app.stride.zone/integrations

Here you will see a list of all of Stride's integrations that you can interact with using your stTokens. There will show you platforms on which you can borrow and lend as well as providing liquidity. For this tutorial, we will show you how to navigate to Osmosis. Below, you can see the Osmosis integration at the top of the list and a link to take you to their DEX.

Before doing this, you need to connect your wallet.

Once you connect your wallet, you will see your TVL and rewards for each platform your stTokens are on. If you select the drop down arrow next to the "Visit" or "Earn" buttons, you can see the available options for your stTokens.

For Osmosis, some of the available liquidity pools and APRs are shown here. If the stToken is available in your wallet, then you can see "Start earning" below the pool. Otherwise, if you do not have STRD or the relevant stToken, you will see "Get [Token] first." For this tutorial, we will following "Start earning" under the ATOM/stATOM pool.

Get Started: Making a Deposit on Osmosis

Use some of your stToken balance (stATOM for this example) to enter the Osmosis pool:

1. From the Stride integrations page, you can go directly to the integration - you can click on the specific pool you wish to provide liquidity for. Another way to get to Osmosis is to simply visit https://app.osmosis.zone/.
2. Click **'Connect Wallet'** in the top right corner.
3. To deposit your stTokens from Stride to Osmosis, select **"Assets"** from the left-hand column. Then, select **"Deposit."** Type the amount you wish to send from your available balance. Complete the transaction in your wallet.

Providing stToken Liquidity to an Osmosis Pool

After completing the deposit, you can now add your available funds to the pool. Check which pools are incentivized at https://frontier.osmosis.zone/pools to determine how you will earn rewards.
Note: Each pool has a pool ID number.
→ For instance, stATOM uses **Pool ID #803**

1. Select **"Pools"** from the left-hand column. Search for the relevant stToken you are going to use to provide liquidity. If you are on Frontier, make sure your pool is incentivized by ticking "Incentivized" pools. (note: some pools are externally incentivized - this means they can be seen when "All Pools" is ticked as shown here, and will be shown in the externally incentivized list further down the page).

This is the page for the **ATOM/stATOM** pool. You will see the pool's total liquidity, volume, fees, and the APR.

2. Select **"Add Liquidity"** and you will be able to enter the amount of each asset (ATOM and stATOM) to provide to the pool. You have the choice to add an equal amount of each asset when **"All assets"** is ticked. If you select **"Single asset"** then the auto-swap feature will allow you to add just ATOM or stATOM.

   Since for this example there is no ATOM balance in the wallet, we will just add stATOM to the pool by selecting **"Single asset."**

4. Enter the amount you wish to add and select **"Add Liquidity."** Complete the transaction in your wallet.

4. When the transaction is complete, you see the confirmation message on screen. The pool page will now reflect your new balance in the pool, your earnings, and more options for your assets. Scroll down on the page for the next step: bonding.

Bonding

1. Bond your liquidity by clicking the green **'Bond Shares'** button.
2. Select the **14 days** unbonding option.
3. To enter the maximum amount, click on the available balance. To input a different balance, type in the text box. Click **Bond** to approve the transaction.

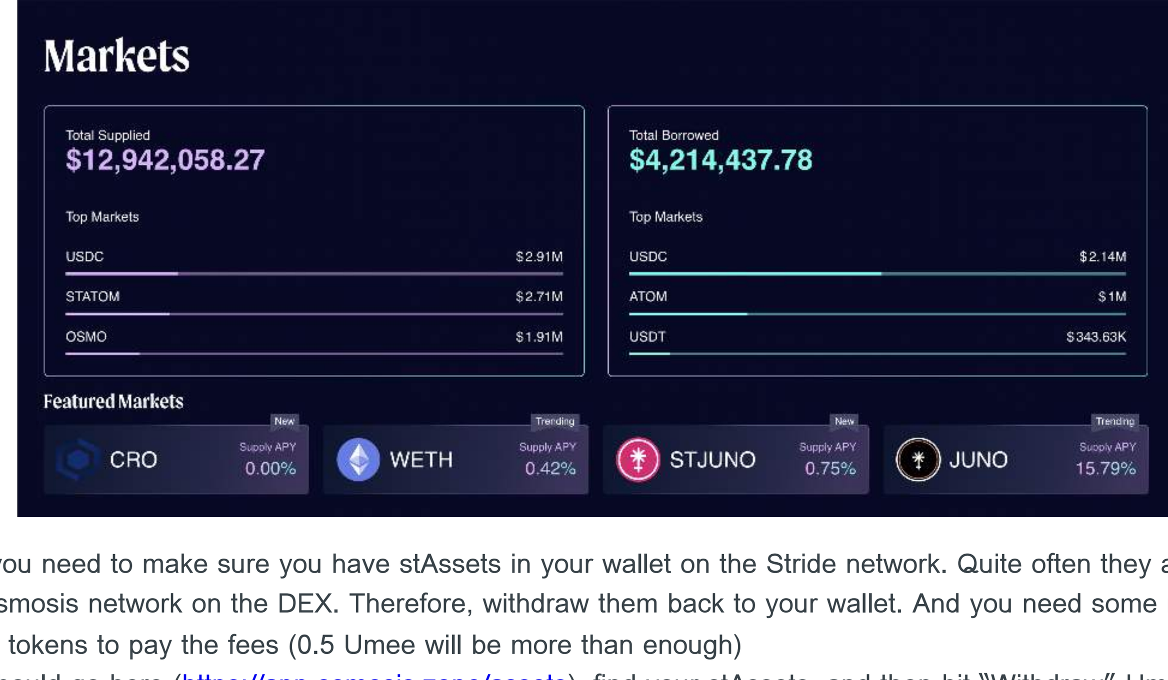
When the transaction is complete, you will see your new balance of bonded liquidity on the pool's info page. You have now successfully provided liquidity using stTokens on Osmosis.

- Pool incentive rewards are paid once per day on Epoch time at 17:15 UTC*

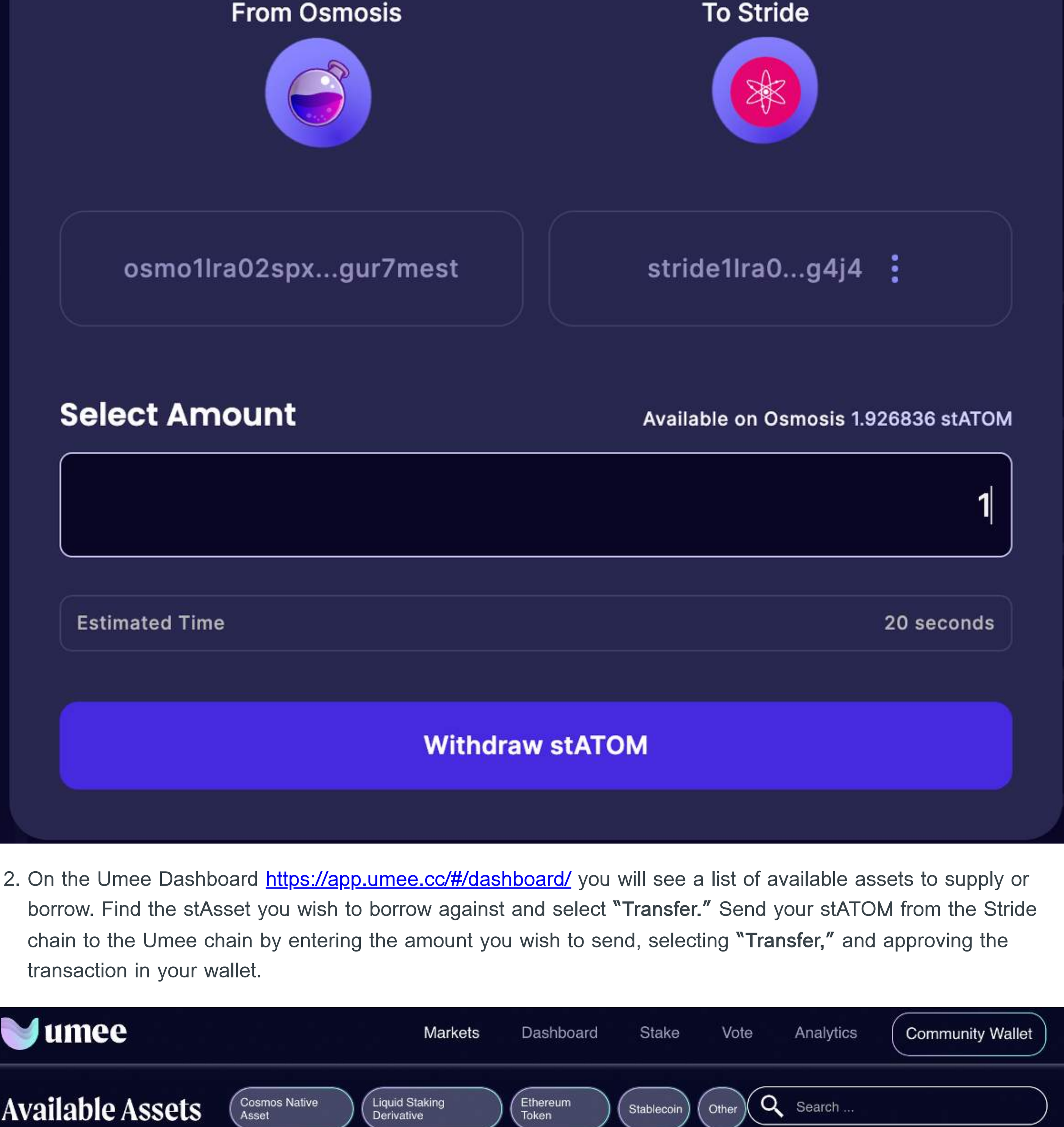
Borrowing against stTokens on Umee

Supplying stTokens on Umee

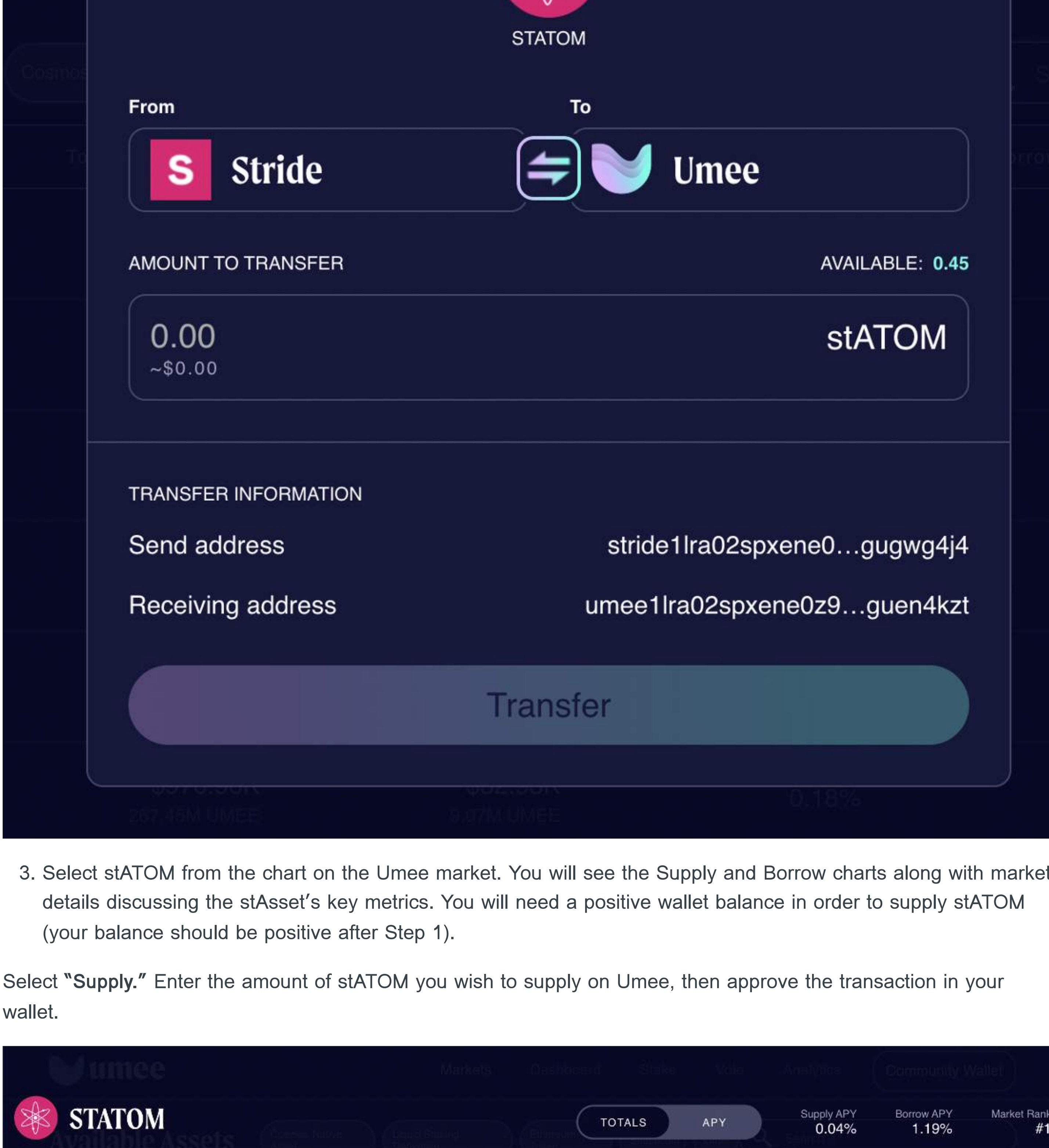
After your stTokens [stAssets] have been transferred to [Umee](), they can be supplied on the [Umee money market]() to earn lending yield and be used as collateral for cross chain borrowing. On the Umee app, you can see a list of featured markets for different assets.



1. First you need to make sure you have stAssets in your wallet on the Stride network. Quite often they are still on the Osmosis network on the DEX. Therefore, withdraw them back to your wallet. And you need some native Umee tokens to pay the fees (0.5 Umee will be more than enough)
   You should go here (https://app.osmosis.zone/assets), find your stAssets, and then hit "Withdraw" Umee tokens can be purchased there as well.

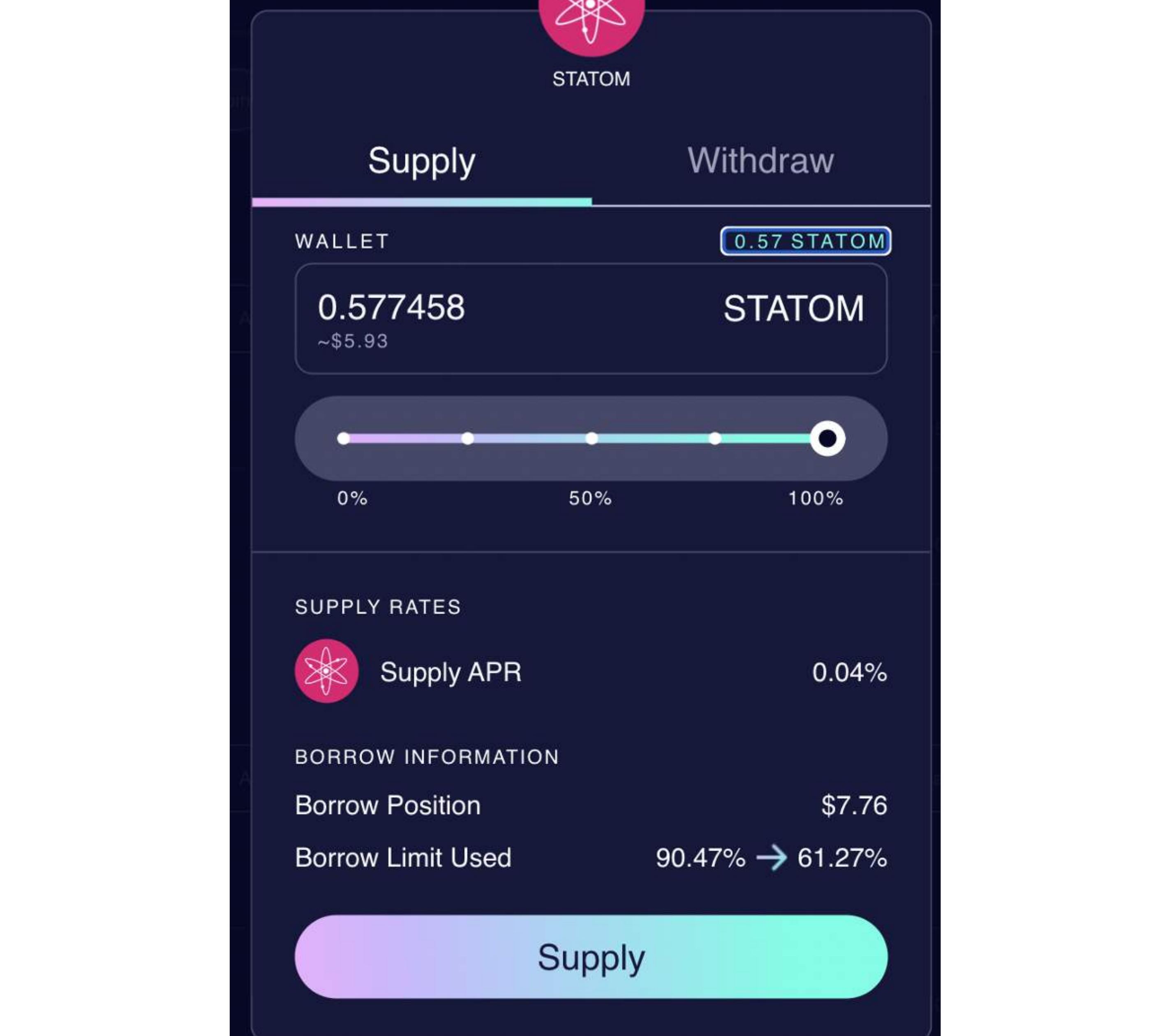


2. On the Umee Dashboard https://app.umee.cc/#/dashboard/ you will see a list of available assets to supply or borrow. Find the stAsset you wish to borrow against and select **"Transfer."** Send your stATOM from the Stride chain to the Umee chain by entering the amount you wish to send, selecting **"Transfer,"** and approving the transaction in your wallet.



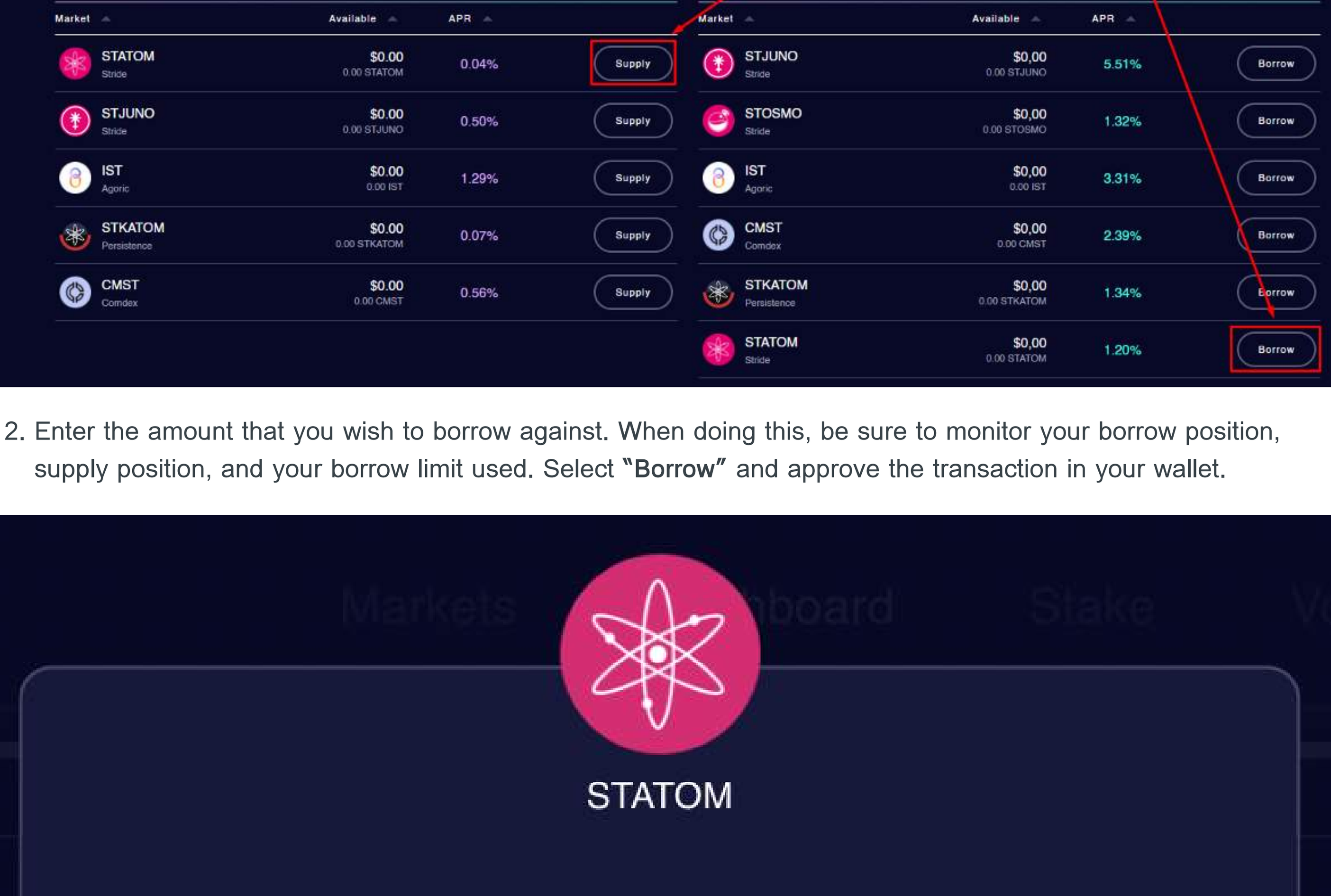
3. Select stATOM from the chart on the Umee market. You will see the Supply and Borrow charts along with market details discussing the stAsset's key metrics. You will need a positive wallet balance in order to supply stATOM (your balance should be positive after Step 1).

Select **"Supply."** Enter the amount of stATOM you wish to supply on Umee, then approve the transaction in your wallet.

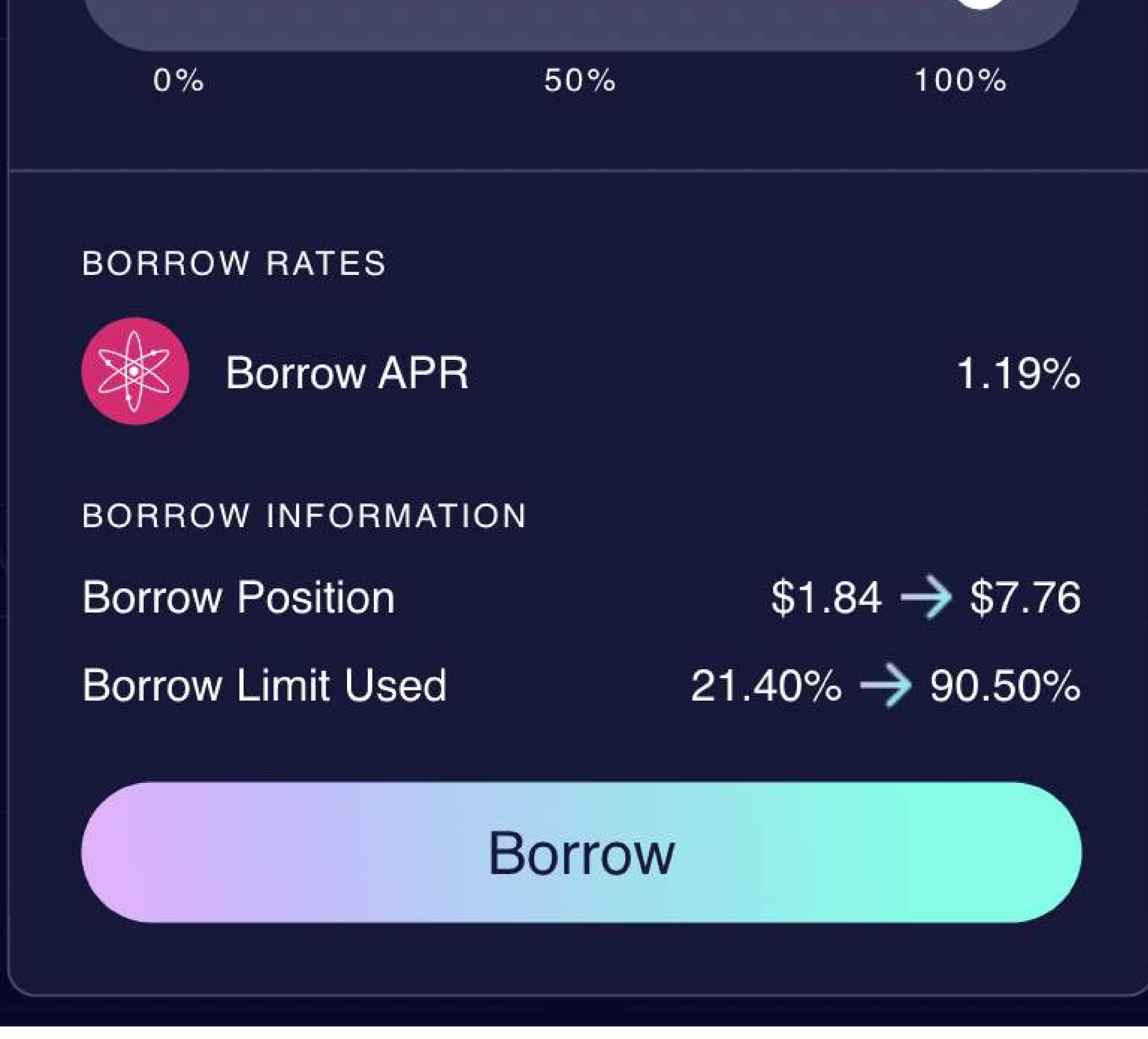


Borrowing Against your stAssets

1. After supplying stATOM as collateral, you are now able to borrow assets on the Umee market. to borrow against your stATOM, select the asset you wish to borrow and hit **"Borrow."** In this case, we will also use stATOM to borrow.



2. Enter the amount that you wish to borrow against. When doing this, be sure to monitor your borrow position, supply position, and your borrow limit used. Select **"Borrow"** and approve the transaction in your wallet.



3. You should now see stATOM in the **"Supply"** and **"Borrow"** sections on the Umee Dashboard, meaning you have both supplied stATOM as collateral and are simultaneously borrowing against it as per the outstanding balance shown on the right.

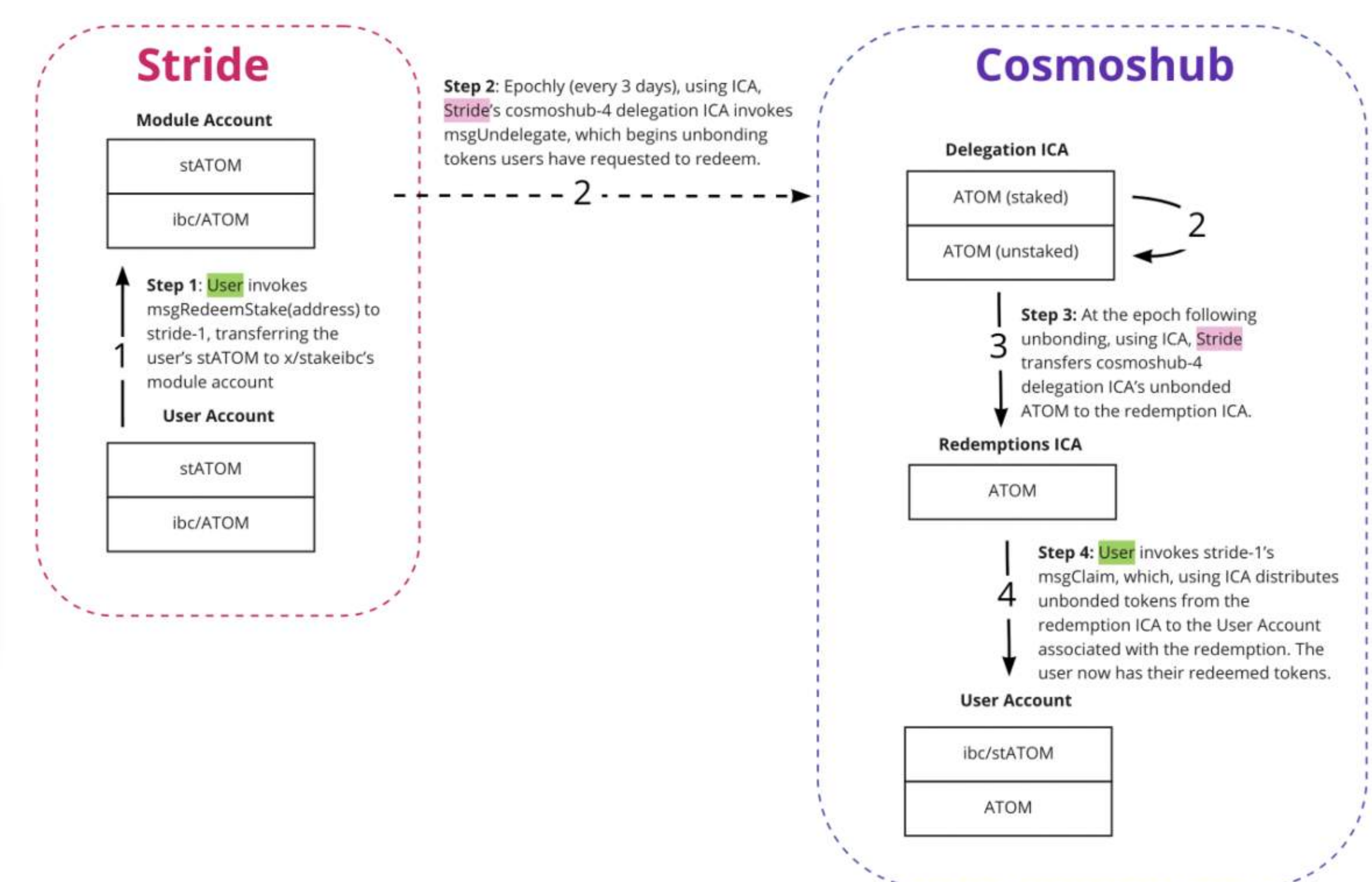
NOTE: Be aware of your borrow limit. Your borrow limit is directly related to how much collateral you have supplied. In this case, since the borrow limit is almost reached, because the borrowed amount is approaching the amount of collateral supplied.

Unstaking Your Tokens

After learning how to liquid stake your tokens, learn how to unstake (unbond) your tokens from Stride.

Stride's Unstaking (Unbonding) Process

At some point after you have staked your tokens and deployed your stTokens in Defi, you will be ready to **unstake**, or **unbond** your staked tokens. In this context, **unstaking or unbonding (these terms can be used interchangeably) means informing the Stride protocol you would like to unlock your tokens, initiating the unbonding period.** The unbonding period is the period of time you must wait before you can access your underlying tokens.



Here's how the process works, using stATOM as an example:

- **Step 0: Liquid stake your ATOM to receive stATOM tokens.**
- **Step 1: Have stATOM in your wallet.** At this point you can use stATOM in DeFi.
- **Step 2: Move your stATOM from your wallet to Stride.**
- **Step 3: Initiate the "Redeem" process (i.e. Unbond).** (In other words, notify that you are ready to redeem your stATOM and initiate the unbonding period).
  - *Example: Vishal wants to unstake 100 stATOM.*
    - Stride records your redemption request and determines how much ATOM your 100 stATOM corresponds to (e.g. 120stATOM).
    - As part of this request, you deposit your 100 stATOM with Stride, and it is burned.
    - Every 4 days, Stride groups all the unbonding requests from all users and processes them.
- **Step 4. Unbonding is initiated.** The Stride blockchain initiates the unbonding process by grouping the records of all of the unbondings on the chain. Unbondings are grouped because Cosmos chains do not allow more than 7 unbondings at a time within a 21 day period. This is a security measure put in place across the Cosmos ecosystem. This does not impact the average user, but it is the reason Stride processes requests every 4 days.
  - The records on the chain show which user the unbonding belongs to, how much ATOM is in the unbonding request, etc. This is lumped into what we call an "epoch" unbonding.
- **When "Unbonding in progress" initiates, it takes 21 days.** When unbonding from Osmosis pools, you still earn external incentives, internal incentives (when Stride incentives a pool), and trade fees during this time. And 21 days later...
- **Step 5: Tokens are moved to the withdrawal account.** Your tokens get automatically moved to a Stride account, whose purpose is to hold unbonded tokens. At this time, the unbonding records (refer to Step 3) get updated on the epoch.
- **Step 6: Claim.** After tokens are fully unbonded, they need to be claimed for tokens to be sent to user accounts. This is currently automated, so that claims are processed automatically. A transaction on the host zone shows the tokens being transfered to the user.

All of the unbonding logic occurs at the "epoch," which is 2PM EST / 19:00 UTC. Note that you cannot claim your tokens until the next 19:00 UTC (so if your tokens are unbonded at 19:05 on one day, they will not be officially claimable until 19:00 the following day - the next epoch).

Note that...
→ Anyone can submit the claim that the tokens are ready to be claimed
→ The claim goes and checks that the record exists
→ Once tokens are claimed, the record is removed
→ All records can be claimed within 30 minutes of being claimable

How to check your unbonding status: https://stride-fleet.main.stridenet.co/api/Stride-Labs/stride/records/epoch_unbonding_record

How to see exact record: https://stride-fleet.main.stridenet.co/api/Stride-Labs/stride/records/user_redemption_record[...]4/31/stride12c34ueuzagjduz5cn4m683kwlt5kq8d2rlwfz4/50

If you need direct support troubleshooting or have a question that is not contained in our FAQ, please join our Discord server and open a Support Ticket.

Getting Started as a Developer

Getting Started as a Developer

Installing Stride

To install the latest version of Stride blockchain node binary, execute the following command on your machine:

```
git clone https://github.com/Stride-Labs/stride
```

Test your installation by navigating to the stride directory and executing:

```
make start-docker build=sgjotr
```

The `build` command installs dependencies, builds, initializes, and starts your blockchain in development mode. You should see logs bring printed to your shell.

If you have any issues with installation, please reach out to our team on Discord.

You can learn more about the install process here or reference these helpful resources:

- Starport
- Tutorials
- Starport docs
- Cosmos SDK docs
- Developer Chat

Developing on Stride

Developers who wish to develop on Stride can easily spin up 3 Stride nodes, 3 Gaia nodes, 1 Hermes relayer and 1 interchain-queries relayer. The Gaia nodes simulate the Cosmos Hub (Gaia) zone in local development node, and the relayers allow Stride zone to interact with that instance of Gaia.

The fastest way to develop on Stride is local development mode.

Set up local development mode

Install the required git submodule dependencies (gaia, hermes, interchain-queries).

```
git submodule update --init
```

Build executables, initialize state, and start the network with

```
make init-local build=sghi
```

You can optionally pass build arguments to specify which binary to rebuild

1. `s` This will re-build the Stride binary (default)
2. `g` This will re-build the Gaia binary
3. `h` This will re-build the Hermes binary
4. `i` This will re-build the ICQ binary

Example: `make init-local build=sg`, this will:

- Rebuild the Stride and Gaia binarys
- Start 1 Stride and 1 Gaia node in the background
- Start Hermes and ICQ Relayers

You can optionally pass `cache=true` to restore the state from a backup instead of re-intializing it.

To bring down the chain, execute:

```
make stop
```

Contributing

Pull requests are welcome. For major changes, please open an issue first to discuss what you would like to change. We also welcome all discussion in #engineering or #questions in our Discord.

Setting Up a Validator

To set up your own validator to support the Stride chain, use the tutorial below to guide you. For information on our inclusive validator sets, see the blog post linked below.

We have a wealth of resources available to any user who wants to set up a validator and connect to Stride. Explore the overview and links attached below for a procedural **overview of running a node.**

If you have questions, you can reach us via email, on Twitter, or at our Discord.
For troubleshooting, we recommend opening a #support-ticket with a Senior Lifeguard on our Discord.

Main requirement for running a node: 4 CPU 32 GB RAM machine *(we have only tested this build on OSX and Linux machine - Windows support coming soon)*

> ### 📘 Installation
>
> See the chain registry for connection info for Stride.
>
> **A quick summary:**
>
> > chain-id = stride-1
> > stride hash = bba22626728961a5a2329031aa34c0140ed76b54
> > genesis file = https://raw.githubusercontent.com/Stride-Labs/testnet/main/mainnet/genesis.json

> ### 📘 Set-Up
>
> **For a step-by-step tutorial, you can use either of these guides:**
>
> - Setting Up a Node
> - How to Set Up a Validator
>
> **Recommended:**
>
> - running a setup with a Sentry node and signed using Horcrux.
> - minimum 8 CPU 64 GB RAM machines

- Stride's Inclusive Validator Sets
- How to Set Up a Validator

Readings: ICS + ICA

For seasoned Cosmonauts who want to contribute as thought leaders on higher level topics, start by exploring the current discourse about ICS, ICA, and other Cosmos innovations. Then, take your thoughts to our community discussion forums in Discord, Twitter, Reddit, and Telegram.

Note that all materials listed are externally sourced.

Interchain Security

What is ICS? Cosmoverse (video)
ICS Explained - Frens Val. (video)
ICS Overview - Coinbase
Interchain Security in Cosmos
Importance of ICS in Cosmos

Interchain Accounts

ICA and UX in Cosmos - Coinbase
ICA and Interoperability
Github - ICA demo
Theta Upgrade/ICA Explained (video)

Query Stride's Redemption Rate

What is this?

Partners integrating with Stride sometimes need to query the rate at which Stride protocol redeems stATOM for ATOM (the "Redemption Rate"). The Stride blockchain exposes this rate via RPC and API endpoints. It can also be recomputed manually.

(Simple) Query the redemption rate directly

You can query the redemption rate directly through REST or the RPC (via the CLI).

- API/REST
  - Show all the host zones, which contain redemption rate fields: `curl -basic https://stride-api.polkachu.com/Stride-Labs/stride/stakeibc/host_zone`
    - Look for the host_zone with your desired chain_id (e.g. for ATOM, look for cosmoshub-4), then choose the field "redemption_rate".
- CLI
  - Show all the host zones, which contain redemption rate fields: `strided-local q stakeibc list-host-zone --node https://stride-rpc.polkachu.com:443`
    - Look for the host_zone with your desired chain_id (e.g. for ATOM, look for cosmoshub-4), then choose the field "redemption_rate".
  - Show a single redemption rate (example for cosmoshub-4): `strided q stakeibc show-host-zone cosmoshub-4 --node https://stride-rpc.polkachu.com:443 | grep redemption_rate: | tail -n 1`

If you'd like a reliable endpoint to query, we recommend Polkachu's. We can provide our team endpoint too, if you'd like.

(Advanced) Recompute the redemption rate manually

You can also reconstruct the redemption rate from Stride's internal record keeping. First, query the relevant internal record keeping, then apply the same formula Stride's beginBlocker logic uses to update the redemption rate. This will yield a redemptionRate that's slightly more updated, relative to that you could query using the previous approach.

- **Step 1: Query Stride's internal record keeping and calculate native token balance**
  - To obtain records for a particular host zone (e.g. cosmoshub-4), query with:
    - API: `curl -basic https://stride-fleet.main.stridenet.co/api/Stride-Labs/stride/records/deposit_record`
    - RPC: `strided q records list-deposit-record --node https://stride-rpc.polkachu.com:443`
  - Compute the inputs to the redemption rate formula:
    - Undelegated record balance:
      - Sum the "amount" field across all records with the proper chain_id having status either DELEGATION_QUEUE or DELEGATION_IN_PROGRESS
    - Module account balance
      - Sum the "amount" field across all records with the proper chain_id having status either TRANSFER_QUEUE or TRANSFER_IN_PROGRESS
    - Staked balance
      - Parse the relevant host zone's `stakedBal` field from `curl -basic https://stride-api.polkachu.com/Stride-Labs/stride/stakeibc/host_zone`
- **Step 2: Calculate the st-token supply (e.g. statom)**: Parse the relevant denom (e.g. stATOM) `strided q bank total --node https://stride-rpc.polkachu.com:443`
- **Step 3: Apply the Redemption Rate formula**
  The formula is `(undelegated record balance + module account balance + staked balance)/(st-token supply)`
  - You can inspect the Stride protocol's logic for updating the redemption rate using this formula.

Integrate Liquid Staking

Using the `x/autopilot` module, protocols can offer Stride liquid staking easily within their own dApp. Users who visit the partner protocol's frontend only need to sign a single transaction to liquid stake from any chain in Cosmos.

Autopilot abstracts away the IBC bridging step so that a user can sign a single ibc-transfer to liquid stake. The user describes their intent within the IBC memo field, where they specify the action they wish to perform via Stride (in this case, that's `LiquidStake`).

Leap Wallet is an early adopter who offers liquid staking directly in their dashboard (check it out).

To enable liquid staking in *your* dApp, prompt an ibc-transfer to Stride with a memo in the following format:

Go

```
{
    "autopilot": {
          "receiver": "strideXXX",
          "stakeibc": {
               "action": "LiquidStake",
          }
    }
}
```

For more details on Autopilot, please see the full documentation.

Branding Resources

Stride's media kit is available at the following link: Media Kit

Module Overviews

Stride implements the following custom modules

`stakeibc` - Manages minting and burning of stTokens, staking and unstaking of native tokens across chains.
`icacallbacks` - Callbacks for interchain accounts.
`records` - IBC middleware wrapping the transfer module, does record keeping on IBC transfers and ICA calls
`claim` - airdrop logic for Stride's rolling, task-based airdrop
`interchainquery` - Issues queries between IBC chains, verifies state proof and executes callbacks.
`epochs` - Makes on-chain timers which other modules can execute code during.
`mint` - Controls token supply emissions, and what modules they are directed to.

Attribution

Stride is proud to be an open-source project, and we welcome all other projects to use our repo. We use modules from the cosmos-sdk and other open source projects.

We operate under the Apache 2.0 License, and have used the following modules from fellow Cosmos projects. Huge thank you to these projects!

We use the following modules from Osmosis provided under this License:

```
x/epochs
x/mint
```

We use the following module (marketed as public infra) from Quicksilver provided under this License:

```
x/interchainqueries
```

Relevant licenses with full attribution can be found in the relevant repo subdirectories.

StakeIBC

The StakeIBC Module

The StakeIBC Module contains Stride's main app logic:

- it exposes core liquid staking entry points to the user (liquid staking and redeeming)
- it executes automated beginBlocker and endBlocker logic to stake funds on relevant host zones using Interchain Accounts
- it handles registering new host zones and adjusting host zone validator sets and weights
- it defines Stride's core data structures (e.g. hostZone)
- it defines all the callbacks used when issuing Interchain Account logic

Nearly all of Stride's functionality is built using interchain accounts (ICAs), which are a new functionality in Cosmos, and a critical component of IBC. ICAs allow accounts on Zone A to be controlled by Zone B. ICAs communicate with one another using Interchain Queries (ICQs), which involve Zone A querying Zone B for relevant information.

Two Zones communicate via a connection and channel. All communications between the Controller Zone (the chain that is querying) and the Host Zone (the chain that is being queried) is done through a dedicated IBC channel between the two chains, which is opened the first time the two chains interact.

For context, ICS standards define that each channel is associated with a particular connection, and a connection may have any number of associated channels.

Params

```
DepositInterval (default uint64 = 1)
DelegateInterval (default uint64 = 1)
ReinvestInterval (default uint64 = 1)
RewardsInterval (default uint64 = 1)
RedemptionRateInterval (default uint64 = 1)
StrideCommission (default uint64 = 10)
ICATimeoutNanos(default uint64 = 600000000000)
BufferSize (default uint64 = 5)
IbcTimeoutBlocks (default uint64 = 300)
FeeTransferTimeoutNanos (default uint64 = 1800000000000)
DefaultMinRedemptionRateThreshold (default uint64 = 90)
DefaultMaxRedemptionRateThreshold (default uint64 = 150)
MaxStakeICACallsPerEpoch (default uint64 = 100)
IBCTransferTimeoutNanos (default uint64 = 1800000000000)
MinRedemptionRates (default uint64 = 90)
MaxRedemptionRates (default uint64 = 150)
ValidatorSlashQueryThreshold (default uint64 = 1)
```

Keeper functions

- `LiquidStake()`
- `RedeemStake()`
- `ClaimUndelegatedTokens()`
- `RebalanceValidators()`
- `AddValidators()`
- `ChangeValidatorWeight()`
- `DeleteValidator()`
- `RegisterHostZone()`
- `ClearBalance()`
- `RestoreInterchainAccount()`
- `UpdateValidatorSharesExchRate()`

State

Callbacks

- `SplitDelegation`
- `DelegateCallback`
- `ClaimCallback`
- `ReinvestCallback`
- `UndelegateCallback`
- `RedemptionCallback`
- `Rebalancing`
- `RebalanceCallback`

HostZone

- `HostZone`
- `ICAAccount`
- `MinValidatorRequirements`

Host Zone Validators

- `Validator`
- `ValidatorExchangeRate`

Misc

- `GenesisState`
- `EpochTracker`
- `Delegation`

Governance

- `AddValidatorsProposal`

Queries

- `QueryInterchainAccountFromAddress`
- `QueryParams`
- `QueryGetValidators`
- `QueryGetHostZone`
- `QueryAllHostZone`
- `QueryModuleAddress`
- `QueryGetEpochTracker`
- `QueryAllEpochTracker`
- `QueryGetNextPacketSequence`

Events

`stakeibc` module emits the following events:

Type: Attribute Key → Attribute Value

registerHostZone: module → stakeibc
registerHostZone: connectionId → connectionId
registerHostZone: chainId → chainId
submitHostZoneUnbonding: hostZone → chainId
submitHostZoneUnbonding: newAmountUnbonding → totalAmtToUnbond
stakeExistingDepositsOnHostZone: hostZone → chainId
stakeExistingDepositsOnHostZone: newAmountStaked → amount
onAckPacket (IBC): module → moduleName
onAckPacket (IBC): ack → ackInfo

Records

The Records Module

The records module handles record keeping and accounting for the Stride blockchain.

It is IBC middleware. IBC middleware wraps core IBC modules and other middlewares. Specifically, the records module adds a middleware stack to `app.go` with the following structure: `records -> transfer`. All ibc packets routed to the `transfer` module will first pass through `records`, where we can apply custom logic (record keeping) before passing messages to the underlying `transfer` module.

Note:

- The middleware stack is added in `app.go`
- The custom handler logic is added in `ibc_module.go` by implementing the IBCModule interface

Keeper functions

Deposit Records

- `GetDepositRecordCount()`
- `SetDepositRecordCount()`
- `AppendDepositRecord()`
- `SetDepositRecord()`
- `GetDepositRecord()`
- `RemoveDepositRecord()`
- `GetAllDepositRecord()`
- `GetTransferDepositRecordByEpochAndChain()`

Epoch Unbonding Records

- `SetEpochUnbondingRecord()`
- `GetEpochUnbondingRecord()`
- `RemoveEpochUnbondingRecord()`
- `GetAllEpochUnbondingRecord()`
- `GetAllPreviousEpochUnbondingRecords()`
- `GetHostZoneUnbondingByChainId()`
- `AddHostZoneToEpochUnbondingRecord()`
- `SetHostZoneUnbondings()`

User Redemption Records

- `SetUserRedemptionRecord()`
- `GetUserRedemptionRecord()`
- `RemoveUserRedemptionRecord()`
- `GetAllUserRedemptionRecord()`
- `IterateUserRedemptionRecords()`

State

Callbacks

- `TransferCallback`

Genesis

- `UserRedemptionRecord`
- `Params`
- `RecordsPacketData`
- `NoData`
- `DepositRecord`
- `HostZoneUnbonding`
- `EpochUnbondingRecord`
- `GenesisState`

Queries

- `Params`
- `GetDepositRecord`
- `AllDepositRecord`
- `GetUserRedemptionRecord`
- `AllUserRedemptionRecord`
- `AllUserRedemptionRecordForUser`
- `GetEpochUnbondingRecord`
- `AllEpochUnbondingRecord`

Events

The `records` module emits does not currently emit any events.

Claim

The Claim Module

Users are required participate in core network activities to claim their airdrop. An Airdrop recipient is given 20% of the airdrop amount which is not in vesting, and then they have to perform the following activities to get the rest:

- 20% vesting over 3 months by staking
- 60% vesting over 3 months by liquid staking

These claimable assets 'expire' if not claimed. Users have three months (`AirdropDuration`) to claim their full airdrop amount. After three months from launch, all unclaimed tokens get sent to the community pool. At initialization, module stores all airdrop users with amounts from genesis inside KVStore. Airdrop users are eligible to claim their vesting or free amount only once in the initial period of 3 months. After the initial period, users can claim tokens monthly.

Actions

There are 2 types of actions, each of which release another 50% of the airdrop allocation.
The 2 actions are as follows:

```golang
ActionLiquidStake  Action = 0
ActionDelegateStake Action = 1
```

These actions are monitored by registring claim **hooks** to the stakeibc, and staking modules.
This means that when you perform an action, the claims module will immediately unlock those coins if they are applicable.
These actions can be performed in any order.

The code is structured by separating out a segment of the tokens as "claimable", indexed by each action type.
So if Alice delegates tokens, the claims module will move the 50% of the claimables associated with staking to her liquid balance.
If she delegates again, there will not be additional tokens given, as the relevant action has already been performed.
Every action must be performed to claim the full amount.

ClaimRecords

A claim record is a struct that contains data about the claims process of each airdrop recipient.

It contains an address, the initial claimable airdrop amount, and an array of bools representing
whether each action has been completed. The position in the array refers to enum number of the action.

So for example, `[true, true]` means that `ActionLiquidStake` and `ActionDelegateStake` are completed.

```golang
type ClaimRecord struct {
 // address of claim user
 Address string
 // weight that represent the portion from total allocation
 Weight sdk.Dec
 // true if action is completed
 // index of bool in array refers to action enum #
 ActionCompleted []bool
}
```

A Note on Address Mappings

When an airdrop is created, we call `LoadAllocationData` to load the airdrop data from the allocations file.
This will apply `utils.ConvertAddressToStrideAddress` on each of those addresses, and then store those with the `ClaimRecords`.
For an airdrop to, say, the Cosmos Hub, this will be the proper Stride address associated with that account.
`claim` state will only ever store this Stride address.

However, for zones with a different coin type, *this will be an incorrect Stride address*. This should not cause any issues though,
as this Stride address will be unusable.

In order to claim that airdrop, the user will have to verify that they own the corresponding Evmos address. When the user tries to verify,
we call `utils.ConvertAddressToStrideAddress` on that address, and verify it gives the same "incorrect" Stride address from earlier.
Through this, we can confirm that the user owns the Evmos address.
We then replace the Stride address with a "correct" one that the user verifies they own.

Params

The airdrop logic has 4 parameters:

```golang
type Params struct {
    // Time that marks the beginning of the airdrop disbursal,
    // should be set to chain launch time.
    AirdropStartTime   time.Time
    AirdropDuration time.Duration
    // denom of claimable asset
    ClaimDenom string
    // address of distributor account
    DistributorAddress string
}
```

Keeper functions

Claim keeper module provides utility functions to manage epochs.

```Go
  GetModuleAccountAddress(ctx sdk.Context) sdk.AccAddress
  GetDistributorAccountBalance(ctx sdk.Context) sdk.Coin
  EndAirdrop(ctx sdk.Context) error
  GetClaimRecord(ctx sdk.Context, addr sdk.AccAddress) (types.ClaimRecord, error)
  GetClaimRecords(ctx sdk.Context) []types.ClaimRecord
  SetClaimRecord(ctx sdk.Context, claimRecord types.ClaimRecord) error
  SetClaimRecords(ctx sdk.Context, claimRecords []types.ClaimRecord) error
  GetClaimableAmountForAction(ctx sdk.Context, addr sdk.AccAddress, action types.Action, includeClaimed
  GetUserTotalClaimable(ctx sdk.Context, addr sdk.AccAddress) (sdk.Coins, error)
  ClaimCoinsForAction(ctx sdk.Context, addr sdk.AccAddress, action types.Action) (sdk.Coins, error)
  clearInitialClaimables(ctx sdk.Context)
  fundRemainingsToCommunity(ctx sdk.Context) error
```

React Hooks

The claim module reacts on the following hooks, executed in external modules.

20% of airdrop is sent to a vesting account when `staking.AfterDelegationModified` hook is triggered.
20% of airdrop is sent to a vesting account when `stakeibc.AfterLiquidStake` hook is triggered.

Once the airdrop is claimed for a specific hook type, it can't be claimed again.

Claim Records

```protobuf
// A Claim Records is the metadata of claim data per address
message ClaimRecord {
  // address of claim user
  string address = 1 [ (gogoproto.moretags) = "yaml:\"address\"" ];

  // weight that represent the portion from total allocations
  double weight = 2;

  // true if action is completed
  // index of bool in array refers to action enum #
  repeated bool action_completed = 3 [
    (gogoproto.moretags) = "yaml:\"action_completed\""
  ];
}
```

When a user get airdrop for his/her action, claim record is created to prevent duplicated actions on future actions.

State

```protobuf
message GenesisState {
  // params defines all the parameters of the module.
  Params params = 2 [
    (gogoproto.moretags) = "yaml:\"params\"",
    (gogoproto.nullable) = false
  ];

  // list of claim records, one for every airdrop recipient
  repeated ClaimRecord claim_records = 3 [
    (gogoproto.moretags) = "yaml:\"claim_records\"",
    (gogoproto.nullable) = false
  ];
}
```

Claim module's state consists of `params`, and `claim_records`.

Claim module provides below params

```protobuf
// Params defines the claim module's parameters.
message Params {
  google.protobuf.Timestamp airdrop_start_time = 1 [
    (gogoproto.stdtime) = true,
    (gogoproto.nullable) = false,
    (gogoproto.moretags) = "yaml:\"airdrop_start_time\""
  ];
  google.protobuf.Timestamp airdrop_duration = 2 [
    (gogoproto.nullable) = false,
    (gogoproto.stdduration) = true,
    (gogoproto.jsontag) = "airdrop_duration,omitempty",
    (gogoproto.moretags) = "yaml:\"airdrop_duration\""
  ];
  // denom of claimable asset
  string claim_denom = 3;
  // airdrop distribution account
  string distributor_address = 4;
}
```

1. `airdrop_start_time` refers to the time when user can start to claim airdrop.
2. `airdrop_duration` refers to the duration from start time to end time.
3. `claim_denom` refers to the denomination of claiming tokens. As a default, it's `ustrd`.
4. `distributor_address` refers to the address of distribution account.

Queries

GRPC queries

Claim module provides below GRPC queries to query claim status

```protobuf
service Query {
  rpc DistributorAccountBalance(QueryDistributorAccountBalanceRequest) returns (QueryDistributorAccount
  rpc Params(QueryParamsRequest) returns (QueryParamsResponse) {}
  rpc ClaimRecord(QueryClaimRecordRequest) returns (QueryClaimRecordResponse) {}
  rpc ClaimableForAction(QueryClaimableForActionRequest) returns (QueryClaimableForActionResponse) {}
  rpc TotalClaimable(QueryTotalClaimableRequest) returns (QueryTotalClaimableResponse) {}
  rpc ClaimStatus(QueryClaimStatusRequest) returns (QueryClaimStatusResponse) {}
  rpc ClaimMetadata(QueryClaimMetadataRequest) returns (QueryClaimMetadataResponse) {}
}
```

CLI commands

For the following commands, you can change `$(strided keys show -a {your key name})` with the address directly.

Query the claim record for a given address

```Shell
strided query claim claim-record $(strided keys show -a {your key name})
```

Query the claimable amount that would be earned if a specific action is completed right now.

```Shell
strided query claim claimable-for-action $(strided keys show -a {your key name}) ActionAddLiquidity
```

Query the total claimable amount that would be earned if all remaining actions were completed right now.

```Shell
strided query claim total-claimable $(strided keys show -a {your key name}) ActionAddLiquidity
```

Query claim status, across all claims, for an address. Returns a list of `ClaimStatus` structs.

```
message ClaimStatus {
  string airdrop_identifier = 1;
  bool claimed = 2;
}
```

```Shell
strided query claim claim-status $(strided keys show -a {your key name})
```

Query claim metadata, across all claims. Returns a `ClaimMetadata` struct, which contains data about the status of each claim.

```
message ClaimMetadata {
  string airdrop_identifier = 1;
  string current_round = 2;
  google.protobuf.Timestamp current_round_start = 3;
  google.protobuf.Timestamp current_round_end = 4;
}
```

```Shell
strided query claim claim-metadata
```

Events

`claim` module emits the following events at the time of hooks:

Type	Attribute Key	Attribute Value
claim	sender	{receiver}
claim	amount	{claim_amount}

Icacallbacks

The ICACallbacks Module

Add `icacallbacks` module. Interchain accounts are very useful, but ICA calls triggered by automated logic on Stride are limited in functionality and difficult to work with due to a lack of callbacks. Most of Stride's interchain account logic is triggered epochly from the `BeginBlocker`, and state updates on Stride must be made after ICA calls are issued, based on the success / failure of those calls.

The challenges faced before creating the icacallbacks modules were:
(1) Messages must be handled one-off (by matching on message type) - really what we want to do is update state in acks based on the *transaction* sent
(2) Message responses are almost always empty, e.g. `MsgDelegateResponse`, which leads to
(3) Matching processed messages to state on the controller is very challenging (sometimes impossible) based on the attributes of the message sent. For example, given the following type

```
type Gift struct {
    from string
    to string
    amount int
    reason string
}
```

two ICA bank sends associated with gifts that have the same `from`, `to` and `amount` but *different* `reasons` are indistinguishable today in acks.

`icacontroller` solves the issues as follows

- Callbacks can be registered to process data associated with a particular IBC packet, per module (solves 1)
- ICA auth modules can store callback data using `icacallbacks`, passing in both a callback and args
- Arguments to the callback can be (un)marshaled and contain arbitrary keys, allowing for updates to data on the controller chain based on the success / failure of the ICA call (solves 2 / 3)

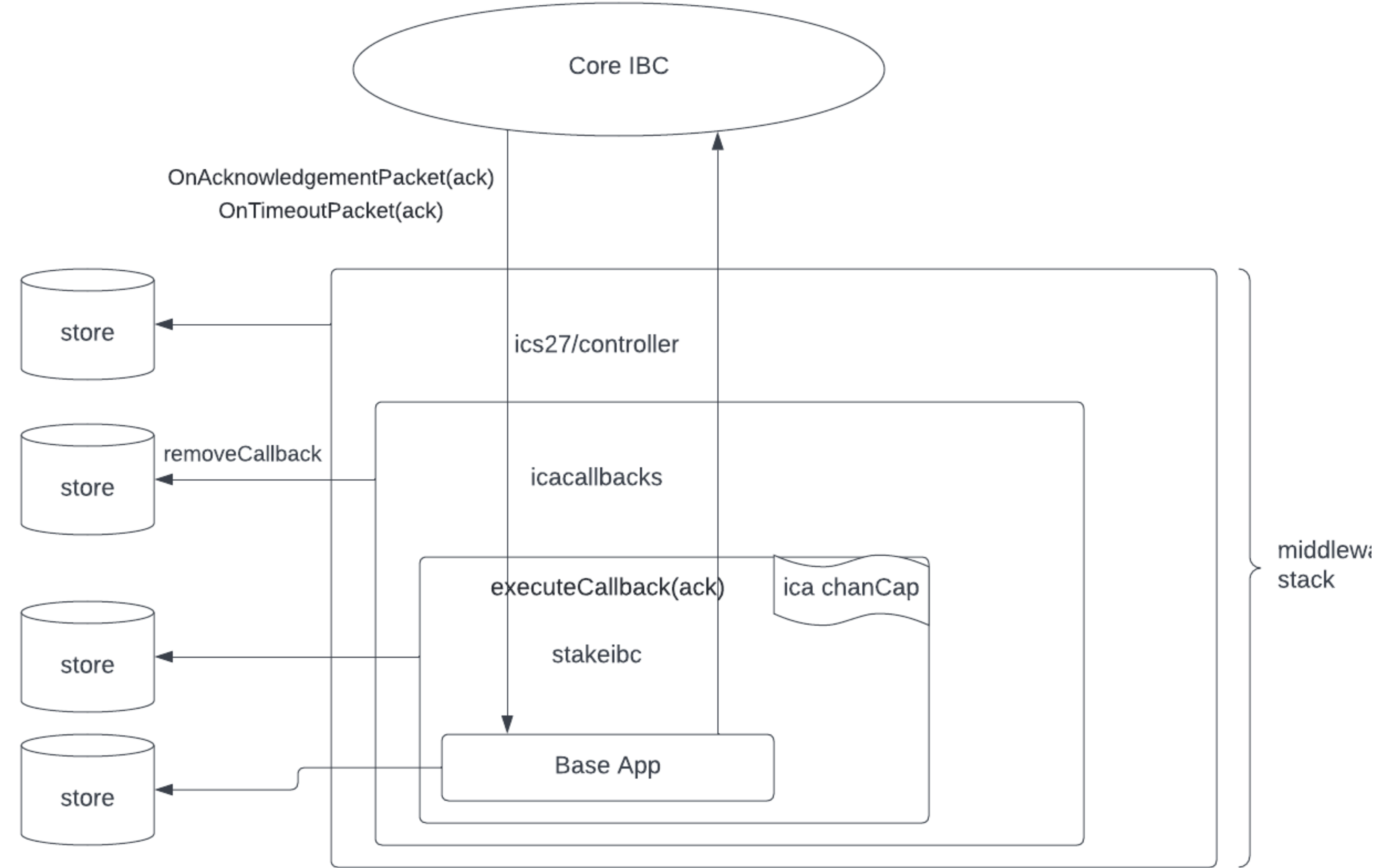
Technical notes

- Callbacks are uniquely identifiable through `portId/channelId/sequence` keys
- Only modules that have registered callbacks can invoke them
- `icacallbacks` doesn't have a message server / handler (it can only be called by other modules)
- `icacallbacks` does authentication by fetching the module associated with a packet (containing the registered callbacks) by calling `ChannelKeeper.LookupModuleByChannel` (it's permissioned at the module level)
- `icacallbacks` is an interchain account auth module, although it's possible this design could be generalized to work with other IBC modules
- in case of a timeout, callbacks are still executed with the ack set to an empty byte array
- We're using protos to serialize / deserialize callback arguments

The flow to add callbacks is to call `ICACallbacksKeeper.SetCallbackData` after sending an IBC transaction. When the ack returns

- the callback is fetched using the callback key
- the module is fetched using the portId / channelId
  and the callback is invoked and deleted.

The middleware structure is as follows



Invariants

- `portId, channelId` pair map to a unique module (important for fetching the correct `CallbackHandler` from a received packet)
- callback ids are unique within modules (they don't have to be unique between modules, because `CallICACallback` is scoped to a module)

Keeper functions

- `CallRegisteredICACallback()`: invokes the relevant callback asssociated with an ICA

State

- `CallbackData`: stores the callback type, arguments and associated packet
- `CallbackHandler`
- `Callbacks`
- `Callback`

Events

The `icacallbacks` module does not currently emit any events.

Interchainquery

Interchain Query

Abstract

Stride uses interchain queries and interchain accounts to perform multichain liquid staking. The `interchainquery` module creates a framework that allows other modules to query other appchains using IBC. The `interchainquery` module is used to make bank balance ICQ queries to withdrawal account every N. The callback triggers ICA bank sends for 90% of the rewards to the delegation account and 10% to the stride hostzone revenue account. The ICA bank send logic is in x/stakeibc/keeper/callbacks.go.

Contents

1. Concepts
2. State
3. Events
4. Keeper
5. Msgs
6. Queries

State

The `interchainquery` module keeps `Query` objects and modifies the information from query to query, as defined in `proto/interchainquery/v1/genesis.proto`

InterchainQuery information type

`Query` has information types that pertain to the query itself. `Query` keeps the following:

1. `id` : query identification string.
2. `connection_id` : id of the connection between the controller and host chain.
3. `chain_id` : id of the queried chain.
4. `query_type` : type of interchain query (e.g. bank store query)
5. `request_data` : serialized request information (e.g. the address with which to query)
6. `callback_module` : name of the module that will handle the callback
7. `callback_id` : ID for the function that will be called after the response is returned
8. `callback_data` : optional serialized data associated with the callback
9. `timeout_policy` : specifies how to handle a timeout (fail the query, retry the query, or execute the callback with a timeout)
10. `timeout_duration` : the relative time from the current block with which the query should timeout
11. `timeout_timestamp` : the absolute time at which the query times out
12. `request_sent` : boolean indicating whether the query event has been emitted (and can be identified by a relayer)
13. `submission_height` : the light client hight of the queried chain at the time of query submission

`DataPoint` has information types that pertain to the data that is queried. `DataPoint` keeps the following:

1. `id` keeps the identification string of the datapoint
2. `remote_height` keeps the block height of the queried chain
3. `local_height` keeps the block height of the querying chain
4. `value` keeps the bytecode value of the data retrieved by the Query

Events

The `interchainquery` module emits an event at the end of every `stride_epoch` s (e.g. 15 minutes on local testnet).

The purpose of this event is to send interchainqueries that query data about staking rewards, which Stride uses to reinvest (aka autocompound) staking rewards.

```go
event := sdk.NewEvent(
  sdk.EventTypeMessage,
  sdk.NewAttribute(sdk.AttributeKeyModule, types.AttributeValueCategory),
  sdk.NewAttribute(sdk.AttributeKeyAction, types.AttributeValueQuery),
  sdk.NewAttribute(types.AttributeKeyQueryId, queryInfo.Id),
  sdk.NewAttribute(types.AttributeKeyChainId, queryInfo.ChainId),
  sdk.NewAttribute(types.AttributeKeyConnectionId, queryInfo.ConnectionId),
  sdk.NewAttribute(types.AttributeKeyType, queryInfo.QueryType),
  sdk.NewAttribute(types.AttributeKeyHeight, "0"),
  sdk.NewAttribute(types.AttributeKeyRequest, hex.EncodeToString(queryInfo.Request)),
)
```

Keeper

Keeper Functions

`interchainquery/keeper/` module provides utility functions to manage ICQs

```go
// GetQuery returns query
GetQuery(ctx sdk.Context, id string) (types.Query, bool)
// SetQuery set query info
SetQuery(ctx sdk.Context, query types.Query)
// DeleteQuery delete query info
DeleteQuery(ctx sdk.Context, id string)
// IterateQueries iterate through queries
IterateQueries(ctx sdk.Context, fn func(index int64, queryInfo types.Query) (stop bool))
// AllQueries returns every queryInfo in the store
AllQueries(ctx sdk.Context) []types.Query
```

Msgs

```protobuf
// SubmitQueryResponse is used to return the query response back to Stride
message MsgSubmitQueryResponse {
  string chain_id = 1;
  string query_id = 2;
  bytes result = 3;
  tendermint.crypto.ProofOps proof_ops = 4;
  int64 height = 5;
  string from_address = 6;
}
```

Queries

```protobuf
// Query PendingQueries lists all queries that have been requested (i.e. emitted)
//  but have not had a response submitted yet
message QueryPendingQueriesRequest {}
```

Epochs

Epochs

Abstract

While using the SDK, we often want to run certain code periodically. The `epochs` module allows other modules to be configured such that they are signaled once every period. So another module can specify it wants to execute code once a week, starting at UTC-time = x. `epochs` creates a generalized epoch interface to other modules so that they can easily be signalled upon such events.

Contents

1. Concepts
2. State
3. Events
4. Keeper
5. Hooks
6. Queries
7. Future Improvements

Concepts

Epochs are on-chain timers that have timer ticks at specific time intervals, triggering the execution of certain logic that is constrained by a specific epoch. The purpose of the `epochs` module is to provide a generalized epoch interface to other modules so that they can easily implement epochs without keeping their own code for epochs.

Every epoch has a unique identifier. Every epoch will have a start time, and an end time, where `end_time = start_time + duration`.
When an epoch triggers the execution of code, that code is executed at the first block whose blocktime is greater than `end_time`. It follows that the `start_time` of the following epoch will be the `end_time` of the previous epoch.

Stride uses three epoch identifiers as found in `x/epochs/genesis.go`

1. `DAY_EPOCH`: this identifies an epoch that lasts 24 hours.
2. `STRIDE_EPOCH`: this identifies an epoch that lasts 5 minutes on local mode tesnet (although this may be changed) and longer on public testnet and mainnet, and is used in the `x/stakeibc/` module as a time interval in accordance with which the Stride app chain performs certain functions, such as autocompound stakig rewards.

State

The `epochs` module keeps `EpochInfo` objects and modifies the information as epoch info changes.
Epochs are initialized as part of genesis initialization, and modified on begin blockers or end blockers.

Epoch information type

```protobuf
message EpochInfo {
    string identifier = 1;
    google.protobuf.Timestamp start_time = 2 [
        (gogoproto.stdtime) = true,
        (gogoproto.nullable) = false,
        (gogoproto.moretags) = "yaml:\"start_time\""
    ];
    google.protobuf.Duration duration = 3 [
        (gogoproto.nullable) = false,
        (gogoproto.stdduration) = true,
        (gogoproto.jsontag) = "duration,omitempty",
        (gogoproto.moretags) = "yaml:\"duration\""
    ];
    int64 current_epoch = 4;
    google.protobuf.Timestamp current_epoch_start_time = 5 [
        (gogoproto.stdtime) = true,
        (gogoproto.nullable) = false,
        (gogoproto.moretags) = "yaml:\"current_epoch_start_time\""
    ];
    bool epoch_counting_started = 6;
    reserved 7;
    int64 current_epoch_start_height = 8;
}
```

`EpochInfo` keeps `identifier`, `start_time`, `duration`, `current_epoch`, `current_epoch_start_time`, `epoch_counting_started`, `current_epoch_start_height`.

1. `identifier` keeps epoch identification string.
2. `start_time` keeps epoch counting start time, if block time passes `start_time`, `epoch_counting_started` is set.
3. `duration` keeps target epoch duration.
4. `current_epoch` keeps current active epoch number.
5. `current_epoch_start_time` keeps the start time of current epoch.
6. `epoch_number` is counted only when `epoch_counting_started` flag is set.
7. `current_epoch_start_height` keeps the start block height of current epoch.

Events

The `epochs` module emits the following events:

BeginBlocker

Type	Attribute Key	Attribute Value
epoch_start	epoch_number	{epoch_number}
epoch_start	start_time	{start_time}

EndBlocker

Type	Attribute Key	Attribute Value
epoch_end	epoch_number	{epoch_number}

Keeper

Keeper Functions

`epochs/keeper/` module provides utility functions to manage epochs.

```go
// Keeper is the interface for lockup module keeper
type Keeper interface {
  // GetEpochInfo returns epoch info by identifier
  GetEpochInfo(ctx sdk.Context, identifier string) types.EpochInfo
  // SetEpochInfo set epoch info
  SetEpochInfo(ctx sdk.Context, epoch types.EpochInfo)
  // DeleteEpochInfo delete epoch info
  DeleteEpochInfo(ctx sdk.Context, identifier string)
  // IterateEpochInfo iterate through epochs
  IterateEpochInfo(ctx sdk.Context, fn func(index int64, epochInfo types.EpochInfo) (stop bool))
  // Get all epoch infos
  AllEpochInfos(ctx sdk.Context) []types.EpochInfo
}
```

Hooks

```go
  // the first block whose timestamp is after the duration is counted as the end of the epoch
  AfterEpochEnd(ctx sdk.Context, epochIdentifier string, epochNumber int64)
  // new epoch is next block of epoch end block
  BeforeEpochStart(ctx sdk.Context, epochIdentifier string, epochNumber int64)
```

The `BeforeEpochStart` hook does different things depending on the identifier.

If in a `day` identifier it:

1. begins unbondings
2. sweeps unbonded tokens to the redemption account
3. cleans up old records
4. creates empty epoch unbonding records for the next day

If in a `stride_epoch` identifier it: 5. creates and deposits records on each host zone 6. sets withdrawal addresses 7. updates redemption rates (if the epoch coincides with the correct interval) 8. processes `TRANSFER_QUEUE` deposit records to the delegation Interchain Account (if the epoch coincides with the correct interval) 9. processes `DELEGATION_QUEUE` deposit records to the delegation Interchain Account (if the epoch coincides with the correct interval) 10. Query the rewards account using interchain queries, with the transfer callback to a delegation account as a staked record (if at proper interval)

How modules receive hooks

On the hook receiver functions of other modules, they need to filter `epochIdentifier` and execute for only a specific `epochIdentifier`.
Filtering `epochIdentifier` could be in `Params` of other modules so that they can be modified by governance.
Governance can change an epoch from `week` to `day` as needed.

Queries

`epochs` module provides the below queries to check the module's state

```protobuf
service Query {
  // EpochInfos provide running epochInfos
  rpc EpochInfos(QueryEpochsInfoRequest) returns (QueryEpochsInfoResponse) {}
  // CurrentEpoch provide current epoch of specified identifier
  rpc CurrentEpoch(QueryCurrentEpochRequest) returns (QueryCurrentEpochResponse) {}
}
```

Future Improvements

Lack point using this module

In current design each epoch should be at least 2 blocks as start block should be different from endblock.
Because of this, each epoch time will be `max(blocks_time x 2, epoch_duration)`.
If epoch_duration is set to `1s`, and `block_time` is `5s`, actual epoch time should be `10s`.
We definitely recommend configure epoch_duration as more than 2x block_time, to use this module correctly.
If you enforce to set it to 1s, it's same as 10s - could make module logic invalid.

TODO for postlaunch: We should see if we can architect things such that the receiver doesn't have to do this filtering, and the epochs module would pre-filter for them.

Block-time drifts problem

This implementation has block time drift based on block time.
For instance, we have an epoch of 100 units that ends at t=100, if we have a block at t=97 and a block at t=104 and t=110, this epoch ends at t=104.
And new epoch start at t=110. There are time drifts here, for around 1-2 blocks time.
It will slow down epochs.

It's going to slow down epoch by 10-20s per week when epoch duration is 1 week. This should be resolved after launch.

Mint

The Mint Module

The `x/mint` module mints tokens at the end of epochs.

The `x/distribution` module is responsible for allocating tokens to stakers, community pool, etc.

The mint module uses time basis epochs from the `x/epochs` module.

The `x/mint` module is designed by Osmosis. It is used to handle the regular printing of new tokens within a chain. Its core function is to:

- Mint new tokens once per epoch (default one week)
- Have a "Reductioning factor" every period, which reduces the amount of rewards per epoch.
  (default: period is 3 years, where a year is 52 epochs. The next period's rewards are 2/3 of the prior period's rewards)

Params

Minting params are held in the global params store.

```go
type Params struct {
    MintDenom               string                  // type of coin to mint
    GenesisEpochProvisions  sdk.Dec                 // initial epoch provisions at genesis
    EpochIdentifier         string                  // identifier of epoch
    ReductionPeriodInEpochs int64                   // number of epochs between reward reductions
    ReductionFactor         sdk.Dec                 // reduction multiplier to execute on each period
 DistributionProportions DistributionProportions // distribution_proportions defines the proportion of
 WeightedDeveloperRewardsReceivers    []WeightedAddress // address to receive developer rewards
 MintingRewardsDistributionStartEpoch int64             // start epoch to distribute minting rewards
}
```

The minting module contains the following parameters:

Key	Type	Example
mint_denom	string	"uosmo"
genesis_epoch_provisions	string (dec)	"500000000"
epoch_identifier	string	"weekly"
reduction_period_in_epochs	int64	156
reduction_factor	string (dec)	"0.6666666666666"
distribution_proportions.staking	string (dec)	"0.4"
distribution_proportions.pool_incentives	string (dec)	"0.3"
distribution_proportions.developer_rewards	string (dec)	"0.2"
distribution_proportions.community_pool	string (dec)	"0.1"
weighted_developer_rewards_receivers	array	[{"address": "osmoxx", "weight": "1"}]
minting_rewards_distribution_start_epoch	int64	10

EpochProvision

Calculate the provisions generated for each epoch based on current epoch provisions. The provisions are then minted by the `mint` module's `ModuleMinterAccount`. These rewards are transferred to a `FeeCollector`, which handles distributing the rewards per the chains needs. (See TODO.md for details) This fee collector is specified as the `auth` module's `FeeCollector` `ModuleAccount`.

```go
func (m Minter) EpochProvision(params Params) sdk.Coin {
    provisionAmt := m.EpochProvisions.QuoInt(sdkmath.NewInt(int64(params.EpochsPerYear)))
    return sdk.NewCoin(params.MintDenom, provisionAmt.TruncateInt())
}
```

Notes

1. `mint_denom` defines denom for minting token - uosmo
2. `genesis_epoch_provisions` provides minting tokens per epoch at genesis.
3. `epoch_identifier` defines the epoch identifier to be used for mint module e.g. "weekly"
4. `reduction_period_in_epochs` defines the number of epochs to pass to reduce mint amount
5. `reduction_factor` defines the reduction factor of tokens at every `reduction_period_in_epochs`
6. `distribution_proportions` defines distribution rules for minted tokens, when developer rewards address is empty, it distribute tokens to community pool.
7. `weighted_developer_rewards_receivers` provides the addresses that receives developer rewards by weight
8. `minting_rewards_distribution_start_epoch` defines the start epoch of minting to make sure minting start after initial pools are set

Begin-Epoch

Minting parameters are recalculated and inflation
paid at the beginning of each epoch. An epoch is signalled by x/epochs

NextEpochProvisions

The target epoch provision is recalculated on each reduction period (default 3 years).
At the time of reduction, the current provision is multiplied by reduction factor (default `2/3`),
to calculate the provisions for the next epoch. Consequently, the rewards of the next period
will be lowered by `1 - reduction factor`.

```go
func (m Minter) NextEpochProvisions(params Params) sdk.Dec {
    return m.EpochProvisions.Mul(params.ReductionFactor)
}
```

Reductioning factor

This is a generalization over the Bitcoin style halvenings.
Every year, the amount of rewards issued per week will reduce by a governance specified factor, instead of a fixed `1/2`.
So `RewardsPerEpochNextPeriod = ReductionFactor * CurrentRewardsPerEpoch)`.
When `ReductionFactor = 1/2`, the Bitcoin halvenings are recreated.
We default to having a reduction factor of `2/3`, and thus reduce rewards at the end of every year by `33%`.

The implication of this is that the total supply is finite, according to the following formula:

$$Total\ Supply = InitialSupply + EpochsPerPeriod * \frac{InitialRewardsPerEpoch}{1 - ReductionFactor} $$

Events

The minting module emits the following events:

End of Epoch

Type	Attribute Key	Attribute Value
mint	epoch_number	{epochNumber}
mint	epoch_provisions	{epochProvisions}
mint	amount	{amount}

Minter

The minter is a space for holding current rewards information.

```go
type Minter struct {
    EpochProvisions sdk.Dec   // Rewards for the current epoch
}
```