What is Superfluid?

Superfluid is a revolutionary asset streaming protocol that brings subscriptions, salaries, vesting, and rewards to DAOs and crypto-native businesses worldwide. This is made possible by the protocol's smart contract framework which introduces the Super Token - an extension to the ERC20 standard that enables digital value to be moved in entirely novel ways.

The Superfluid framework & Super Token standard can be used to add dynamic balances to tokens on chain, describing cash flows and executing them automatically over time in a non-interactive way. Any token can be transferred in Superfluid streams, which are programmable, composable, and modular. No capital is locked up, and all inflows and outflows are netted in real-time at every block without consuming any gas. The code is fully open source, while the protocol is non-custodial and permissionless.

Token-Centric Framework

The **Super Token** is an extension of the ERC20 standard that has additional powerful value-movement abilities beyond the basic "transfer". We call these abilities Super Agreements.

Currently there are two additional Super Agreements:

What is Superfluid?

Superfluid is a revolutionary asset streaming protocol that brings subscriptions, salaries, vesting, and rewards to DAOs and crypto-native businesses worldwide. This is made possible by the protocol's smart contract framework which introduces the Super Token - an extension to the ERC20 standard that enables digital value to be moved in entirely novel ways.

The Superfluid framework & Super Token standard can be used to add dynamic balances to tokens on chain, describing cash flows and executing them automatically over time in a non-interactive way. Any token can be transferred in Superfluid streams, which are programmable, composable, and modular. No capital is locked up, and all inflows and outflows are netted in real-time at every block without consuming any gas. The code is fully open source, while the protocol is non-custodial and permissionless.

Token-Centric Framework

The **Super Token** is an extension of the ERC20 standard that has additional powerful value-movement abilities beyond the basic "transfer". We call these abilities Super Agreements.

Currently there are two additional Super Agreements:

🚬 **"Money Streams" - Constant Flow Agreement (CFA)**: move tokens in constant-rate by-the-second streams between accounts with no capital lock up.

🎆 **"Instant Distributions" - Instant Distribution Agreement (IDA)**: assign proportions to many receiving accounts and lump-distribute tokens to them on the basis of the set proportions

These agreements modify the balance of Super Tokens just like basic transfers would! There's no claiming/settling/etc. involved - Superfluid instantly settles everything within the Super Token balance.

Composable

ERC20 tokens are the most composable Web3 primitive we know of - you can borrow against them, exchange them, stake them, the list is endless. Since all Superfluid boils down to is an extended ERC20 token, its composability is limitless as well.

You can do all those basic ERC20 things in addition to the many possibilities introduced by money streams and instant distributions. Imagine a contract where you stake an NFT and receive a stream of Super Tokens straight to your wallet in return (no need to claim). Or, imagine a dollar-cost averaging protocol that scalably distributes Super Tokens straight to investor wallets with a fixed gas cost.

Unprecedented mechanisms like this are made possible by the composability of Super Tokens.

Previous ←

Next - Protocol Overview →

claiming/settling/etc. involved - Superfluid instantly settles everything within the Super Token balance.

Composable

ERC20 tokens are the most composable Web3 primitive we know of - you can borrow against them, exchange them, stake them, the list is endless. Since all Superfluid boils down to is an extended ERC20 token, its composability is limitless as well.

You can do all those basic ERC20 things in addition to the many possibilities introduced by money streams and instant distributions. Imagine a contract where you stake an NFT and receive a stream of Super Tokens straight to your wallet in return (no need to claim). Or, imagine a dollar-cost averaging protocol that scalably distributes Super Tokens straight to investor wallets with a fixed gas cost.

Unprecedented mechanisms like this are made possible by the composability of Super Tokens.

← Previous
Superfluid

Next - Protocol Overview
In-Depth Overview →

claiming/settling/etc. involved - Superfluid instantly settles everything within the Super Token balance.

Composable

ERC20 tokens are the most composable Web3 primitive we know of - you can borrow against them, exchange them, stake them, the list is endless. Since all Superfluid boils down to is an extended ERC20 token, its composability is limitless as well.

You can do all those basic ERC20 things in addition to the many possibilities introduced by money streams and instant distributions. Imagine a contract where you stake an NFT and receive a stream of Super Tokens straight to your wallet in return (no need to claim). Or, imagine a dollar-cost averaging protocol that scalably distributes Super Tokens straight to investor wallets with a fixed gas cost.

Unprecedented mechanisms like this are made possible by the composability of Super Tokens.

← Previous
Superfluid

Next - Protocol Overview
In-Depth Overview →

claiming/settling/etc. involved - Superfluid instantly settles everything within the Super Token balance.

Composable

ERC20 tokens are the most composable Web3 primitive we know of - you can borrow against them, exchange them, stake them, the list is endless. Since all Superfluid boils down to is an extended ERC20 token, its composability is limitless as well.

You can do all those basic ERC20 things in addition to the many possibilities introduced by money streams and instant distributions. Imagine a contract where you stake an NFT and receive a stream of Super Tokens straight to your wallet in return (no need to claim). Or, imagine a dollar-cost averaging protocol that scalably distributes Super Tokens straight to investor wallets with a fixed gas cost.

Unprecedented mechanisms like this are made possible by the composability of Super Tokens.

← Previous
Superfluid

Next - Protocol Overview
In-Depth Overview →

Super App Testing on Mainnet Forks

Learn how to test Super Apps on Hardhat mainnet forks

> This section is based off the work of omnifient, a valued community member r🚀

In order for a Super App's callbacks to work, it needs to be registered with the Superfluid host.

On testnets, this is done permissionlessly, but on a mainnet it requires a registration key which is permissioned by Superfluid governance. See the constructor below 👇

```
constructor(
    ISuperfluid host,
    string memory registrationKey
) {

    uint256 configWord = SuperAppDefinitions.APP_LEVEL_FINAL |
        SuperAppDefinitions.BEFORE_AGREEMENT_CREATED_NOOP |
        SuperAppDefinitions.BEFORE_AGREEMENT_UPDATED_NOOP |
```

```
        SuperAppDefinitions.BEFORE_AGREEMENT_TERMINATED_NOOP;

    if (bytes(registrationKey).length > 0) {
        // works on MAINNET
        host.registerAppWithKey(configWord, registrationKey);
    } else {
        // works on TESTNET
        host.registerApp(configWord);
    }

}
```

So, in order to test a Super App on a mainnet fork, you are going to need to be able to *simulate the creation of a registration key* so that you can provide it to your Super App upon deployment.

Here's how you do it in Hardhat's Node.js-based environment.

Simulating Creation of a Registration Key

First, get an instance of the Superfluid Host. Here, we're just setting up the instance with the only function we'll be using - `getGovernance`

```
const HOST_ABI = ["function getGovernance() external view returns (address)"];
const HOST_ADDR = "0x3E14dC1b13c488a8d5D310918780c983bD5982E7";

const hostInstance = new ethers.Contract(HOST_ADDR, HOST_ABI, <<providerOrSigner>>);
```

```
const hostInstance = new ethers.Contract(host_ADDR, host_ABI, <<providerOrSigner>>);
```

Next, let's get an instance of the Superfluid Governance contract and its owner's address.

```
const GOV_ABI = [
  "function setConfig(address host, address superToken, bytes32 key, uint256 value) exte
  "function setAppRegistrationKey(address host, address deployer, string memory registra
  "function getConfigAsUint256(address host, address superToken, bytes32 key) external v
  "function verifyAppRegistrationKey(address host, address deployer, string memory regis
  "function owner() public view returns (address)",
];

const govInstance = new ethers.Contract(
  await hostInstance.getGovernance(),
  GOV_ABI,
  <<providerOrSigner>>
);

const govOwnerAddr = await govInstance.owner();
```

Now, impersonate the governance owner!

```
// impersonate the governance owner
await network.provider.request({
  method: "hardhat_impersonateAccount",
```

```
  params: [govOwnerAddr],
});

// give it a lot of ETH (or MATIC because we're on Polygon!)
await network.provider.send("hardhat_setBalance", [
  govOwnerAddr,
  hexValue(parseEther("1000000")),
]);

const govOwnerSigner = await ethers.getSigner(govOwnerAddr);
```

Time to make your registration key

```
// Set your registration key string
const registrationKey = `GM-${Date.now()}`;

// Get a configuration key
const configKey = ethers.utils.keccak256(
  ethers.utils.defaultAbiCoder.encode(
    ["string", "address", "string"],
    [
      "org.superfluid-finance.superfluid.appWhiteListing.registrationKey",
      <<account that will deploy the Super App>>,
      registrationKey
    ]
  )
```

```
);

// Set it in Governance - now your registrationKey is good to go!
let tx = await govInstance.setConfig(
  HOST_ADDR,
  "0x0000000000000000000000000000000000000000",
  configKey,
  Math.floor(Date.now() / 1000) + 3600 * 24 * 180 // 180 day expiration for key
);
await tx.wait();
```

Finally, deploy the Super App with the `registrationKey` you've permissioned!

```
const superAppFactory = await hre.ethers.getContractFactory(<<SuperAppContractName>>);
const mySuperApp = await factory.deploy(HOST_ADDR, registrationKey);

await mySuperApp.deployed();
```


← Previous
Hardhat Mainnet Fork Testing

Next - Developers
Subgraph →

```
// Set it in Governance - now your registrationKey is good to go!
let tx = await govInstance.setConfig(
  HOST_ADDR,
  "0x0000000000000000000000000000000000000000",
  configKey,
  Math.floor(Date.now() / 1000) + 3600 * 24 * 180 // 180 day expiration for key
);
await tx.wait();
```

Finally, deploy the Super App with the `registrationKey` you've permissioned!

```
const superAppFactory = await hre.ethers.getContractFactory(<<SuperAppContractName>>);
const mySuperApp = await factory.deploy(HOST_ADDR, registrationKey);

await mySuperApp.deployed();
```


Previous
Hardhat Mainnet Fork Testing

Next - Developers
Subgraph

Subgraph

Hungry for data? Hot n' fresh Superfluid subgraphs ready to consume!

The Graph is the indexing layer of our industry. There is an ocean of open data created by blockchain networks, and the Graph will help you query it. You can use the Graph to get data on activity in the Superfluid ecosystem as well as many other types of on-chain data. You can also use the GraphQL Playground to start getting Superfluid specific data on any network.

URLs

Click a link to start testing queries in the playground:

Network	URL
xDAI	https://console.superfluid.finance/subgraph?_network=xdai
Matic	https://console.superfluid.finance/subgraph?_network=matic
Avalanche	https://console.superfluid.finance/subgraph?

	_network=avalanche-c
Optimism	https://console.superfluid.finance/subgraph?_network=optimism-mainnet
Arbitrum	https://console.superfluid.finance/subgraph?_network=arbitrum-one
Avalanche Fuji	https://console.superfluid.finance/subgraph?_network=avalanche-fuji
Optimism Kovan	https://thegraph.com/hosted-service/subgraph/superfluid-finance/protocol-v1-optimism-kovan
Arbitrum Rinkeby	https://thegraph.com/hosted-service/subgraph/superfluid-finance/protocol-v1-arbitrum-rinkeby
Mumbai	https://console.superfluid.finance/subgraph?_network=mumbai
Goerli	https://console.superfluid.finance/subgraph?_network=goerli
Ropsten	https://thegraph.com/hosted-service/subgraph/superfluid-finance/protocol-v1-ropsten

Alice ("0x658...") and Bob ("0xd66...")

`token` will return the token entity type - with the `symbol` and `id` in this case.

`createdAtTimestamp` will return the timestamp at which the `stream` was created.

`updatedAtTimestamp` will return the timestamp of the last time the stream was updated.

`currentFlowRate` will return the current flow rate of the stream.

`streamedUntilUpdatedAt` will return the amount streamed until `updatedAtTimestamp` / `updatedAtBlock`

> Note: to get the current (dynamic) total amount streamed, you can use the following formula:
> ***streamedUntilUpdatedAt + ((current time in seconds) - updatedAtTimestamp) * currentFlowRate***

```
{
    streams(where:{
          sender: "0x658e1b019f2f30c8089a9ae3ae5820f335bd9ce4"
          receiver: "0xd66e40b0c30595bec72153b502ac1e0c4785991b"
        }
       ) {
        token {
        id
        symbol
      }
            createdAtTimestamp
```

```
            updatedAtTimestamp
            currentFlowRate
            streamedUntilUpdatedAt
        }
    }
```

Get The Most Recently Updated Flows

Here, we can query the FlowUpdatedEvent type and use `first`, `orderBy` and `orderDirection` to get the 10 most recently updated flows.

> Note: as explained in the previous higher order level entities section under the `Stream` type, it's important to understand the difference between a Flow and a Stream. A Flow represents all streaming activity between 2 addresses for a given token, while a Stream may only represent a portion of that flow. A new stream period is created each time the `flowRate` is updated, and a new stream is created each time a stream is terminated and restarted again (with the creation happening upon restarting)

`oldFlowRate` - the flowRate of the previous stream (will return "0" if this is a newly created Flow)

`flowRate` - the flowRate of the current stream

`userData` - user data associated with the update.

`stream` - the Stream type associated with the flow. In our case, the next few items will come from

the Stream type. Several streams may be returned if there have been multiple streams created for this `flow`.

`token` - the `token` used in the updated flow. Here we are returning this token's `symbol`

`sender` - the `sender` of the updated flow. Here we will return this sender's `id` (their address)

`receiver` - the `receiver` of the updated flow. Here we will return this receiver's `id` (their address)

```
{
   flowUpdatedEvents(first: 10, orderBy: timestamp, orderDirection: desc) {
        oldFlowRate
        flowRate
        userData
        stream {
            token {
            symbol
        }
            sender {
                id
        }
            receiver {
                id
        }
      }
  }
```

```
}
```

Get an Account's IDA Index Subscriptions

Here, we can query all of the subscriptions a particular account is subscribed to.

`subscriptions` will return an `IndexSubscription` type for each `Index` that this account is subscribed to

`token` - the `Token` used in each subscription. Here we display the `symbol`.

`id` - the ID of the `IndexSubscription`. This ID is composed of: `subscriberAddress` - `publisherAddress` - `tokenAddress` - `IndexId`

`approved` - a boolean value that represents whether or not the subscription is `approved`. Approved subscriptions don't require subscribers to claim tokens that are distributed from the publisher.

`units` - the total number of units allocated to this account in the index. If `units` is 0, it indicates that the subscription is "deleted". They are no longer subscribed to the index.

`totalAmountReceivedUntilUpdatedAt` - the total amount of tokens that this account has received via this IDA index until `updatedAtTimestamp` / `updatedAtBlock`.

```
{
   accounts(where: {
   #your address below
```

```
    id: "0x..."
    }) {
    subscriptions{
        index {
          token{
            symbol
          }
        }
        id
        approved
        units
        totalAmountReceivedUntilUpdatedAt
    }
  }
}
```

Get Recent Index Information By Querying an Event Type

In this example, we can filter by & return data using a combination of an event type and HOL type. Here, we'll get the 10 most recent indexUpdatedEvents where the newIndexValue is greater than 1000 tokens.

We'll return the following from the Index type

`indexValue` - used to calculate the distribution amount that a user receives when the publisher updates the index value. The distribution amount can be calculated using the following formula:

> (index.indexValue - subscriber.indexValue) * subscriber.units

`totalUnitsPending` - he number of units allocated by the `Index` that are pending. This refers to the current (as of updatedAt) `totalUnitsPending` - not all that has ever been pending.

`totalUnitsApproved` - he number of units allocated by the `Index` that have been approved. This refers to the current (as of updatedAt) `totalUnitsApproved` - not all that has ever been approved.

`totalUnits` - the sum of `totalUnitsPending` and `totalUnitsApproved`.

`token` - the type `Token` used in the index.

`symbol` - the symbol of the `Token` used in the index

```
{
   indexUpdatedEvents (
      first: 10,
      orderBy: timestamp
      orderDirection: desc
      where: {
        indexValue_gt: "1000000000000000000000"
      }) {
          index {
          indexValue
          totalUnitsPending
```

```
            totalUnitsPending
            totalUnitsApproved
            totalUnits
            token {
              symbol
            }
          }
    }
  }
```

Get Aggregate Flow Data For a Given Token

As an example of how to use aggregate entities, let's take a look at a query using the Token Statistic type. In this case, we can see aggregate data for Super DAI on Polygon.

`totalNumberOfActiveStreams` - the total count of active streams using the token.

`totalNumberOfActiveIndexes` - total number of active IDA indexes for the token.

`totalAmountStreamedUntilUpdatedAt` - all-time total amount streamed until the `updatedAtTimestamp` / `updatedAtBlock`.

`totalOutFlowRate` - total outflow rate of the `token` (how much value is being moved).

`totalAmountDistributedUntilUpdatedAt` - all-time total amount distributed until the `updatedAtTimestamp` / `updatedAtBlock`.

```
{
  tokenStatistics(where: {
     #DAIx address on Matic
     id: "0x1305f6b6df9dc47159d12eb7ac2804d4a33173c2"
     })
        {
           totalNumberOfActiveStreams
           totalNumberOfActiveIndexes
           totalAmountStreamedUntilUpdatedAt
           totalOutflowRate
           totalAmountDistributedUntilUpdatedAt
        }
}
```

Get Data On a Specific Account

To query data on a specific account, we can use both the Account entity and the aggregate AccountTokenSnapshot entity. Note: if you're only interested in seeing an account's activity with a particular token, you can query AccountTokenSnapshot to get that data. Otherwise, the `Account` entity will return an `AccountTokenSnapshot` for every token used for each `id` (i.e. for each Account's address).

On the `Account` entity:

`isSuperApp` - returns `true` if the account is a super app, `false` if not.

`inflows` - will return all `Streams` where `Account` is the `sender`.

`outflows` - will return all `Streams` where `Account` is the `receiver`.

On the `AccountTokenSnapshot` entity:

`token` - the address of the token used in streams or subscriptions

`totalNumberOfActiveStreams` - the total number of streams the account is associated with for the given token

`totalNetFlowRate` - the total net flow rate of the `account` for the given token as of `updatedAtTimestamp` / `updatedAtBlock`.

```
{
  accounts(where: {
  #enter an address below
  id: "0x..."
  })
   {
     isSuperApp
     inflows {
      currentFlowRate
      token{
        symbol
      }
      sender {
        id
```

```
      }
    }
     outflows {
      currentFlowRate
      token {
        symbol
      }
      receiver {
        id
      }
    }
     accountTokenSnapshots{
     token {
       id
     }
     totalNumberOfActiveStreams
     totalNetFlowRate
    }
  }
}
```

Your query here!

If you design a useful query, let us know and we will share it here!

Learn more Subgraph Documentation

	finance/protocol-v1-ropsten
Kovan	https://thegraph.com/hosted-service/subgraph/superfluid-finance/protocol-v1-kovan
Rinkeby	https://thegraph.com/hosted-service/subgraph/superfluid-finance/protocol-v1-rinkeby

Resources

- ★ **New to GraphQL required reading** - https://graphql.org/learn/
- **How to make subgraph queries** - https://thegraph.com/docs/en/developer/query-the-graph/
- **Deploy your own subgraph** - https://thegraph.com/docs/en/developer/create-subgraph-hosted/
- **GraphQL Schema for Superfluid** - https://github.com/superfluid-finance/protocol-monorepo/blob/dev/packages/subgraph/schema.graphql
- Apollo Client - for writing queries in your front end - https://www.apollographql.com/docs/

Contributing & Local Deployment

If you'd like to help improve the subgraph, there is a complete development guide available in the

Legacy Subgraph Documentation

Legacy Subgraph URLs

Click a link to start testing queries in the playground:

Network	URL
xDAI	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-xdai
Matic	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-matic
Mumbai	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-mumbai
Goerli	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-goerli
Ropsten	https://thegraph.com/hosted-service/subgraph/superfluid-

Ropsten	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-ropsten
Kovan	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-kovan
Rinkeby	https://thegraph.com/hosted-service/subgraph/superfluid-finance/superfluid-rinkeby

Legacy Schema

The Legacy schema contains fewer entities and some subtle differences in schema. These differences are outlined in the above sections, but you can find the previous schema in full here:

protocol-monorepo/schema.graphql at ethereum-contracts@v1.0.0-rc.7 · superfluid-fi...
GitHub

> NOTE: Please refer to the above section on Breaking Changes as of October 2021, and the section on Streams vs Flows in the above section on the V1 Subgraph documentation. For example, you'll notice in the below example that the `Flow` type is used in a query. This type has

example, you'll notice in the below example that the `Flow` type is used in a query. This type has been deprecated.

How much USDCx has Alice streamed to Bob? [*Legacy Query*]

The Subgraph can only return a flow's `sum` since the last on-chain event (`flowUpdated` , `flowCreated` , or `flowDeleted`). Therefore, if `flowRate` is greater than zero, you will need to calculate the tokens flowed since the last event. Use this equation to get the the total sum of tokens flowed:

$$totalSum = sum + flowRate(currentTimestamp - lastUpdate)$$

```
{
    tokens(where: { symbol: "USDCx" }) {
        flows(
            where: {
                owner: "0x658e1b019f2f30c8089a9ae3ae5820f335bd9ce4"
                recipient: "0xd66e40b0c30595bec72153b502ac1e0c4785991b"
            }
        ) {
            sum
            flowRate
            lastUpdate
        }
    }
}
```

repo:

protocol-monorepo/packages/subgraph at dev · superfluid-finance/protocol-monorepo
GitHub

Helpful Tips

- All addresses (`id`, `underlyingAddress`, etc.) are lowercase (i.e. not checksummed). Be sure to convert addresses to lowercase first before making queries.

Notable Breaking Changes When Migrating From Our Legacy Subgraph to V1 (as of October 2021)

- The field `totalSubscriptions` on the `Index`, `AccountTokenSnapshot` and `TokenStatistic` types is now `totalSubscriptionsWithUnits`.
- The `Subscriber` entity has been changed to `Subscription`. If you have been using the `subscriber` property when querying any events, this has been changed to `subscriptions` everywhere. The only place where `subscriber` is still a field is on the `Subscription` entity as a property of type `Account`.
- The `createdAt` and `updatedAt` fields are now `createdAtTimestamp` and `updatedAtTimestamp` on every type.

Schema

The Superfluid Subgraph consists of many entities that can be queried. You can think of entities as similar to tables within a traditional database. We have added significantly more entities in October 2021 - particularly new event entities and aggregate entities. We'll give you an overview of the subgraph's structure in this post, but you can always dive deeper by looking at the schema inside of our repository.

Event Entities

These entities are for the most part 1-to-1 mappings with the raw events from the contract. However for some of them, we have added additional data. This is noted in the `schema.graphql` file and will be viewable in the playground as well. These entities are created once and never updated. For example, the `FlowUpdatedEvent` type will mirror the properties on the `FlowUpdated` event inside of the Superfluid contracts.

> Note: Each event entity `id` is formulated as follows: `eventName-transactionHash-logIndex`.

> Note: event types in our legacy subgraph did not have 'Event' appended onto the end of their name. In our new v1 implementation, every event type is named 'Event Name' + 'Event.' For example, `FlowUpdated` is now `FlowUpdatedEvent` and `IndexCreated` is now `IndexCreatedEvent`. This schema is applied to every other event type as well.

> `IndexCreatedEvent`. This schema is applied to every other event type as well.

Higher Order Level Entities

Higher order level entities (HOL) are an additional abstraction on top of events. They are not ephemeral in the same way the event entities are. They contain data on an entity over its "lifetime" and therefore may be updated over time. Here is an overview of our Higher Order Level Entities:

`Account`

An entity created for any address which interacts with the Superfluid protocol.

`Token`

An entity created for all 'valid' SuperTokens. Valid tokens are defined as tokens that have set the Superfluid host's value to the correct Superfluid host address within the token contract.

> Note: this network directory is where you can find the address of the host for each network.

`Index`

An entity that represents an `Index` within a Superfluid instant distribution agreement (IDA)

`IndexSubscription`

An entity that contains subscription data for a `subscriber` account of a particular `index`

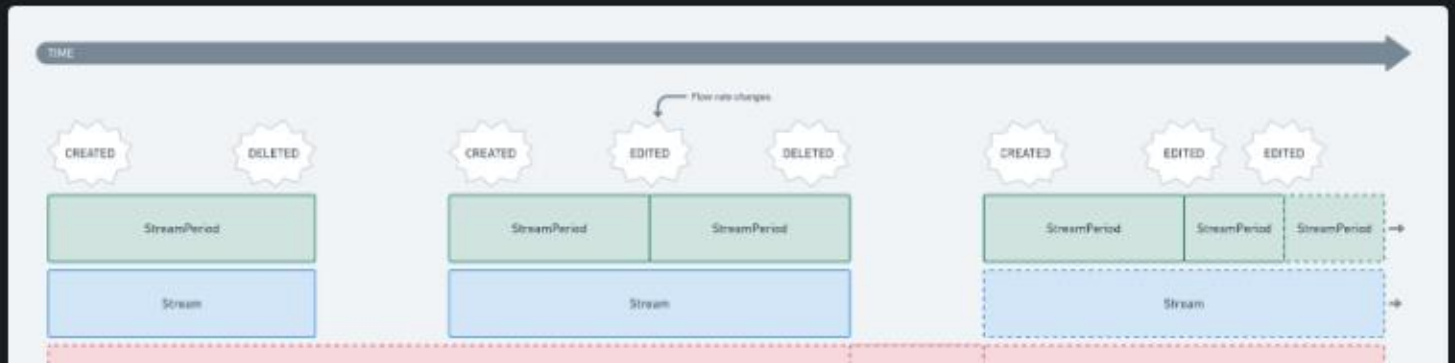
`Stream`

An entity that represents the lifetime of a stream between a `sender` and a `receiver`. An account can create a stream or update its flow rate, but when they delete it, it is considered "dead." The next stream that is created between the same `sender` and `receiver` will generate a new stream entity. Therefore, multiple stream entities can be created between the same `sender` and `receiver`.

`StreamPeriod`

An entity that represents a period of time in a `Stream` with a specific constant flowRate. You can think of it as a portion of a `Stream`. Each time a Stream's `flowRate` is updated, a new `StreamPeriod` will be created.

> Note: the stream type is a replacement for the previously used Flow type (as of October '21). The below image provides a good visual outline of how the `Stream` and `StreamPeriod` types compare to the previously used `Flow` type.



A StreamPeriod vs a Stream vs a Flow

Aggregate Entities

Aggregate entities are exactly what the name suggests - they are entities that store aggregate data. More specifically, we do this at an account-token level and also at a global token level. Here is an overview of our two Aggregate Entities:

`TokenStatistic`

An entity which aggregates all data for a single Token type.

`AccountTokenSnapshot`

An aggregate entity which aggregates data on an account's interaction with a given `token`.

Examples

There are many different ways to get the same data using GraphQL. Here are some examples to help get you started using the Superfluid subgraphs.

Super Token Data

The number of items returned by a query can sometimes be limited. In this example, we are explicitly requesting that up to 100 items should be returned, by using the `first` parameter

> Note: keep in mind that a maximum of 1000 items may be returned in a single query.

```
{
  tokens(first: 100) {
    id
    symbol
    name
    underlyingAddress
  }
}
```

Get All Streams for a Given Account

List all streams that an account is currently receiving. You can swap receiver for sender to get all streams that an account is currently sending.

```graphql
//graphql
query MyQuery {
  streams(where: {receiver: "YOUR_ADDRESS_HERE"}) {
    currentFlowRate
    token {
      symbol
    }
    sender {
      id
    }
    receiver {
      id
    }
  }
}
```

Getting Stream Data Between 2 Parties

The following query will allow us to answer the following questions for an active stream between

Deploying The Framework

Deploying Super Apps and the Superfluid Framework Either Locally or To a New Network

Testing

Truffle/Hardhat

The `@superfluid-finance/ethereum-contracts` package includes helpful scripts for deploying the framework using Truffle + Ganache. These deployment scripts can also be used within a Hardhat environment. You can find examples for writing tests using the Superfluid SDK core in our examples repository:

- Hardhat Example
- Truffle Example

```
//example truffle test setup

const deployFramework = require("@superfluid-finance/ethereum-contracts/scripts/deploy-
```

```
const deployTestToken = require("@superfluid-finance/ethereum-contracts/scripts/deploy-t
const deploySuperToken = require("@superfluid-finance/ethereum-contracts/scripts/deploy-

contract("My Test", accounts => {
    const [admin, bob, carol, dan] = accounts;

    before(async () => {
        await deployFramework(errorHandler, {
            web3,
            from: admin
        });
    });

    beforeEach(async function() {
        await deployTestToken(errorHandler, [":", "fDAI"], {
            web3,
            from: admin
        });
        await deploySuperToken(errorHandler, [":", "fDAI"], {
            web3,
            from: admin
        });
    });

    //...
```

Production

Anyone can deploy the Superfluid Framework contracts to any EVM compatible network using the Superfluid Protocol Deployment Guide. The protocol is live in production on

- Gnosis Chain
- Polygon Network (Matic)
- Optimism
- Arbitrum

More networks are coming soon. Please reach out to us if you want your network added!

Once the contracts are deployed, ownership of the governance contracts should be transferred to a Superfluid-controlled account. This will allow the Superfluid team to manage security upgrades (if necessary), add new Agreements, and improve the core protocol. The governance process will be improved over time, as the community grows and matures.

← Developers - Previous
Reference

Next - Developers
Contract Addresses →

Last modified 9mo ago

Production

Anyone can deploy the Superfluid Framework contracts to any EVM compatible network using the Superfluid Protocol Deployment Guide. The protocol is live in production on

- Gnosis Chain
- Polygon Network (Matic)
- Optimism
- Arbitrum

More networks are coming soon. Please reach out to us if you want your network added!

Once the contracts are deployed, ownership of the governance contracts should be transferred to a Superfluid-controlled account. This will allow the Superfluid team to manage security upgrades (if necessary), add new Agreements, and improve the core protocol. The governance process will be improved over time, as the community grows and matures.

← Developers - Previous
Reference

Next - Developers
Contract Addresses →

Contract Addresses

Superfluid contract addresses for all networks

Note: these addresses can also be found in the Superfluid Console.

Mainnet Networks

Ethereum | Polyg... | Gnosis Ch... | Optimis... | Arbitru... | Avalanc... | BNB Ch... | Ce...

Framework

Resource	Address
Resolver	0xeE4cD028f5fdaAdeA99f8fc38e8bA8A57c90Be53
Host	0x4E583d9390082B65Bef884b629DFA426114CED6d

CFAv1	0x2844c1BBdA121E9E43105630b9C8310e5c72744b
IDAv1	0xbCF9cfA8Da20B591790dF27DE65C1254Bf91563d
SuperTokenFactory	0x0422689cc4087b6B7280e0a7e7F655200ec86Ae1
SuperfluidLoaderV1	0x9775EEBdfF8AE845e7Ad3d1D04B85e6c6d284aCB
TOGA	0x8B5a2CF69a56d7F8Fa027edcA23594cdDF544dDc
Vesting Scheduler	0x39D5cBBa9adEBc25085a3918d36D5325546C001B
CFAV1Forwarder	0xcfA132E353cB4E398080B9700609bb008eceB125

Tokens

Resource	Address

USDCx	0x1BA8603DA702602A8657980e825A6DAa03Dee93a
DAIx	0x4F228bf911ed67730e4B51B1F82AC291B49053ee
ETHx	0xC22BeA0Be9872d8B7B3933CEc70Ece4D53A900da

Test Networks

Mumbai is our encouraged testnet. Goerli is sunsetting, we suggest you refrain from using it.

Mumbai | Optimism Goerli | Arbitrum Goerli | Avalanche Fuji | Goerli

Framework

Resource	Address
Resolver	0x8C54C83FbDe3C59e59dd6E324531FB93d4F504d3
Host	0xEB796bdb90fFA0f28255275e16936D25

Host	d3418603
CFAv1	0x49e565Ed1bdc17F3d220f72DF0857C26FA83F873
IDAv1	0x804348D4960a61f2d5F9ce9103027A3E849E09b8
SuperTokenFactory	0xb798553db6eb3d3c5691237840937014 5e97324b
SuperfluidLoader v1	0x0d56ED56b63382B0FC964490feB9AE438B6B4b79
CFAv1Forwarder	0xcfA132E353cB4E398080B9700609bb008eceB125
Vesting Scheduler	0x2a00b420848D723A74c231B559C117Ee003B1829
Stream Scheduler	0xF18825d412C061aEfEFB4dF46a1c077636dA50bf
Auto Wrap Manager	0x3eAB3c6207F488E475b7955B631B564F0E6317B9
Auto Wrap Strategy	0x544728AFDBeEafBeC9e1329031788edb53017bC4

Tokens

Resource	Address
MATICx	0x96B82B65ACF7072eFEb00502F45757F254c2a0D4
fDAI	0x15F0Ca26781C3852f8166eD2ebce5D18265cceb7
fUSDC	0xbe49ac1EadAc65dccf204D4Df81d650B50122aB2
fTUSD	0xA794C9ee519FD31BbCE643e8D8138f735E97D1DB
fDAIx	0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f
fUSDCx	0x42bb40bF79730451B11f6De1CbA222F17b87Afd7
fTUSDx	0x918E0d5C96cAC79674E2D38066651212be3C9C48

fUSDC	0xbe49ac1EadAc65dccf204D4Df81d650B50122aB2
fTUSD	0xA794C9ee519FD31BbCE643e8D8138f735E97D1DB
fDAIx	0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f
fUSDCx	0x42bb40bF79730451B11f6De1CbA222F17b87Afd7
fTUSDx	0x918E0d5C96cAC79674E2D38066651212be3C9C48

← Previous
Deploying The Framework

Next - Sentinels
Liquidations & TOGA →

Solidity

The Super Token Library allows you to work with money streams in Solidity

SuperTokenV1Library Contract

protocol-monorepo/SuperTokenV1Library.sol at dev · superfluid-finance/protocol-mo...
GitHub

Quickstart Guide

Money Streaming with Solidity 101 | ETHGlobal Guides
ethglobal

Make a contract that streams money in under 10 minutes!

Example Code

super-examples/MoneyRouter.sol at main · superfluid-finance/super-examples
GitHub

Getting Set To Start Streams

Initialize the CFAv1Library in your constructor with the below code.

Initializing the Library

```
// initializing the CFA Library
pragma solidity 0.8.14;

import {
    ISuperfluid
} from "@superfluid-finance/ethereum-contracts/contracts/interfaces/superfluid/ISuperflu

import {
    ISuperToken
} from "@superfluid-finance/ethereum-contracts/contracts/interfaces/superfluid/ISuperToI

import {
    SuperTokenV1Library
} from "@superfluid-finance/ethereum-contracts/contracts/apps/SuperTokenV1Library.sol";
```

```
contract SomeContractWithSuperTokenV1Library {

    using SuperTokenV1Library for ISuperToken;
    ISuperToken public token;

    constructor(ISuperToken _token) {
        token = _token;
    }

    // your contract code here...

}
```

Interacting With The CFA

Read Methods

Write Methods

```
    // your contract code here...

}
```

Interacting With The CFA

Read Methods

Write Methods

Developers - Previous
Money Streaming (CFA)

Next
Read Methods

Read Methods

Method Catalog

```
/** CFA VIEW FUNCTIONS ************************************** */


function getFlowRate(ISuperToken token, address sender, address receiver)
    internal view returns(int96 flowRate);

function getFlowInfo(ISuperToken token, address sender, address receiver)
    internal view
    returns(uint256 lastUpdated, int96 flowRate, uint256 deposit, uint256 owedDeposit);

function getNetFlowRate(ISuperToken token, address account)
    internal view returns (int96 flowRate);

function getNetFlowInfo(ISuperToken token, address account)
    internal view returns (uint256 lastUpdated, int96 flowRate, uint256 deposit, uint25

function getBufferAmountByFlowRate(ISuperToken token, int96 flowRate) internal view
```

```
function getFlowInfo(ISuperToken token, address sender, address receiver)
    internal view
    returns(uint256 lastUpdated, int96 flowRate, uint256 deposit, uint256 owedDeposit);

function getNetFlowRate(ISuperToken token, address account)
    internal view returns (int96 flowRate);

function getNetFlowInfo(ISuperToken token, address account)
    internal view returns (uint256 lastUpdated, int96 flowRate, uint256 deposit, uint25

function getBufferAmountByFlowRate(ISuperToken token, int96 flowRate) internal view
    returns (uint256 bufferAmount);

function getFlowPermissions(ISuperToken token, address sender, address flowOperator)
    internal view
    returns (bool allowCreate, bool allowUpdate, bool allowDelete, int96 flowRateAllowa
```


← Previous
Solidity

Next
getFlowInfo →

Last modified 1mo ago

getFlowInfo

Function Header

```
/**
 * @dev get flow info between two accounts for given token
 * @param token The token used in flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 * @return lastUpdated Timestamp of flow creation or last flowrate change
 * @return flowRate The flow rate
 * @return deposit The amount of deposit the flow
 * @return owedDeposit The amount of owed deposit of the flow
 */
function getFlowInfo(ISuperToken token, address sender, address receiver) internal view
    returns(uint256 lastUpdated, int96 flowRate, uint256 deposit, uint256 owedDeposit)
```

Example Usage

```
 * @param receiver The receiver of the flow
 * @return lastUpdated Timestamp of flow creation or last flowrate change
 * @return flowRate The flow rate
 * @return deposit The amount of deposit the flow
 * @return owedDeposit The amount of owed deposit of the flow
 */
function getFlowInfo(ISuperToken token, address sender, address receiver) internal view
    returns(uint256 lastUpdated, int96 flowRate, uint256 deposit, uint256 owedDeposit)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

Previous
Read Methods

Next
getFlowRate

Last modified 1mo ago

getFlowRate

Function Header

```
/**
 * @dev get flow rate between two accounts for given token
 * @param token The token used in flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 * @return flowRate The flow rate
 */
function getFlowRate(ISuperToken token, address sender, address receiver)
    internal view returns(int96 flowRate)
```

Example Usage

super-examples/projects/flow-splitter/contracts/FlowSplitter.sol at 0b90a34602dc9ca...
GitHub

```
/**
 * @dev get flow rate between two accounts for given token
 * @param token The token used in flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 * @return flowRate The flow rate
 */
function getFlowRate(ISuperToken token, address sender, address receiver)
    internal view returns(int96 flowRate)
```

Example Usage

super-examples/projects/flow-splitter/contracts/FlowSplitter.sol at 0b90a34602dc9ca...
GitHub

← Previous
getFlowInfo

Next
getNetFlowRate →

Last modified 1mo ago

getNetFlowRate

Function Header

```
/**
 * @dev get net flow rate for given account for given token
 * @param token Super token address
 * @param account Account to query
 * @return flowRate The net flow rate of the account
 */
function getNetFlowRate(ISuperToken token, address account)
    internal view returns (int96 flowRate)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

```
/**
 * @dev get net flow rate for given account for given token
 * @param token Super token address
 * @param account Account to query
 * @return flowRate The net flow rate of the account
 */
function getNetFlowRate(ISuperToken token, address account)
    internal view returns (int96 flowRate)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

← Previous
getFlowRate

Next
Write Methods →

Last modified 1mo ago

Write Methods

Method Catalog

```
/** CFA CRUD ************************************ */


function createFlow(ISuperToken token, address receiver, int96 flowRate)
    internal returns (bool);

function createFlow(ISuperToken token, address receiver, int96 flowRate, bytes memory u
    internal returns (bool);

function updateFlow(ISuperToken token, address receiver, int96 flowRate)
    internal returns (bool);

function updateFlow(ISuperToken token, address receiver, int96 flowRate, bytes memory u
    internal returns (bool);

function deleteFlow(ISuperToken token, address sender, address receiver)
    internal returns (bool);
```

```
function deleteFlow(ISuperToken token, address sender, address receiver, bytes memory u
    internal returns (bool);


/** CFA ACL ************************************ */


function setFlowPermissions(
    ISuperToken token,
    address flowOperator,
    bool allowCreate,
    bool allowUpdate,
    bool allowDelete,
    int96 flowRateAllowance
) internal returns (bool);

function setMaxFlowPermissions(
    ISuperToken token,
    address flowOperator
) internal returns (bool);

function revokeFlowPermissions(
    ISuperToken token,
    address flowOperator
) internal returns (bool);

function increaseFlowRateAllowance(ISuperToken token, address flowOperator, int96 added
```

```
    internal returns (bool);

function increaseFlowRateAllowance(
    ISuperToken token,
    address flowOperator,
    int96 addedFlowRateAllowance,
    bytes memory userData
) internal returns (bool);

function decreaseFlowRateAllowance(ISuperToken token, address flowOperator, int96 subtra
    internal returns (bool);

function decreaseFlowRateAllowance(
    ISuperToken token,
    address flowOperator,
    int96 subtractedFlowRateAllowance,
    bytes memory userData
) internal returns (bool);

function setFlowPermissionsWithCtx(
    ISuperToken token,
    address flowOperator,
    bool allowCreate,
    bool allowUpdate,
    bool allowDelete,
    int96 flowRateAllowance,
    bytes memory ctx
```

```
) internal returns (bytes memory newCtx);

function setMaxFlowPermissionsWithCtx(
    ISuperToken token,
    address flowOperator,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function revokeFlowPermissionsWithCtx(
    ISuperToken token,
    address flowOperator,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function createFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate
) internal returns (bool);

function createFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate,
    bytes memory userData
```

```
) internal returns (bool);

function updateFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate
) internal returns (bool);

function updateFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate,
    bytes memory userData
) internal returns (bool);

function deleteFlowFrom(
    ISuperToken token,
    address sender,
    address receiver
) internal returns (bool);

function deleteFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    bytes memory userData
```

```
    bytes memory userData
) internal returns (bool);


/** CFA With CTX FUNCTIONS ********************************** */


function createFlowWithCtx(
    ISuperToken token,
    address receiver,
    int96 flowRate,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function createFlowFromWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function updateFlowWithCtx(
    ISuperToken token,
    address receiver,
    int96 flowRate,
    bytes memory ctx
) internal returns (bytes memory newCtx);
```

```
function updateFlowFromWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function deleteFlowWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function deleteFlowFromWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    bytes memory ctx
) internal returns (bytes memory newCtx);
```

Previous

Next

```
    bytes memory ctx
) internal returns (bytes memory newCtx);

function deleteFlowWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    bytes memory ctx
) internal returns (bytes memory newCtx);

function deleteFlowFromWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    bytes memory ctx
) internal returns (bytes memory newCtx);
```


Previous
getNetFlowRate

Next
createFlow

createFlow

Function Header

```
/**
 * @dev Create flow without userData
 * @param token The token used in flow
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function createFlow(ISuperToken token, address receiver, int96 flowRate)
    internal returns (bool)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

```
/**
 * @dev Create flow without userData
 * @param token The token used in flow
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function createFlow(ISuperToken token, address receiver, int96 flowRate)
    internal returns (bool)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

Previous
Write Methods

Next
updateFlow

Last modified 1mo ago

updateFlow

Function Header

```
/**
 * @dev Update flow without userData
 * @param token The token used in flow
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function updateFlow(ISuperToken token, address receiver, int96 flowRate)
    internal returns (bool)
```

Example Usage

super-examples/projects/borrow-against-salary/contracts/EmploymentLoan.sol at 0b...
GitHub

```
/**
 * @dev Update flow without userData
 * @param token The token used in flow
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function updateFlow(ISuperToken token, address receiver, int96 flowRate)
    internal returns (bool)
```

Example Usage

super-examples/projects/borrow-against-salary/contracts/EmploymentLoan.sol at 0b...
GitHub

Previous
createFlow

Next
deleteFlow

Last modified 1mo ago

deleteFlow

Function Header

```
/**
 * @dev Delete flow without userData
 * @param token The token used in flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 */
function deleteFlow(ISuperToken token, address sender, address receiver)
    internal returns (bool)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

```
/**
 * @dev Delete flow without userData
 * @param token The token used in flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 */
function deleteFlow(ISuperToken token, address sender, address receiver)
    internal returns (bool)
```

Example Usage

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

Previous
updateFlow

Next
setFlowPermissions

Last modified 1mo ago

Quickstart

Find what you need to getting off the ground quick with building on Superfluid

Money Streaming | Distributions

The Constant Flow Agreement (CFA) allows for money streaming. Read this short explainer on how it works and then find the resources you need below!

Money Streaming with SDK Core

- **Set Up**: How to initialize our SDK Core to start conveniently accessing Superfluid functionality
- **Functionality**: See the functions that allow you to manage streams using the SDK Core.
- **Simple Example**: Very simple ReactJS examples showing how to allow a user to create, update, and delete their streams.

Money Streaming with Solidity

- **Set Up**: Set up your smart contract with the SuperTokenV1Library to easily manage money streams in Solidity.
- **Functionality**: See the basic stream management functions of the SuperTokenV1Library
- **ACL**: See how you can provide other addresses with the ability to create, update, and delete streams on your behalf
- **Simple Example**: "Money Router" smart contract example showing the very basics of coding stream management into a smart contract
- **Video Tutorial**: Tutorial demonstrating coding the Money Router contract

Testing With Money Streams

- **Initializing**: How to initialize our SDK Core for your Hardhat testing script to start conveniently accessing Superfluid functionality.
- **Simple Example**: Our "Tradeable Cashflow" example project Hardhat test script shows a lot of the basics.

delete streams on your behalf

- **Simple Example**: "Money Router" smart contract example showing the very basics of coding stream management into a smart contract
- **Video Tutorial**: Tutorial demonstrating coding the Money Router contract

Testing With Money Streams

- **Initializing**: How to initialize our SDK Core for your Hardhat testing script to start conveniently accessing Superfluid functionality.
- **Simple Example**: Our "Tradeable Cashflow" example project Hardhat test script shows a lot of the basics.

← Protocol Overview - Previous
Glossary of Terms

Next - Developers
Super Tokens →

setFlowPermissions

Function Header

```
/**
 * @dev Update permissions for flow operator
 * @param token The token used in flow
 * @param flowOperator The address given flow permissions
 * @param allowCreate creation permissions
 * @param allowCreate update permissions
 * @param allowCreate deletion permissions
 * @param flowRateAllowance The allowance provided to flowOperator
 */
function setFlowPermissions(
    ISuperToken token,
    address flowOperator,
    bool allowCreate,
    bool allowUpdate,
    bool allowDelete,
    int96 flowRateAllowance
) internal returns (bool) {
```

> ⓘ The `flowRateAllowance` works just like allowance for regular ERC20 tokens. See it explained **here**.

Example Usage

```
// Giving Alice permission to only create flows (no update or delete permissions
// of someSuperToken from the contract up to an allowance of 100,000 wei/sec.

someSuperToken.setFlowPermissions(
    alice,
    true,
    false,
    false,
    100000
)
```


← Previous
deleteFlow

Next
setMaxFlowPermissions →

The "FlowRateAllowance" works just like allowance for regular ERC20 tokens. See it explained here.

Example Usage

```
// Giving Alice permission to only create flows (no update or delete permissions
// of someSuperToken from the contract up to an allowance of 100,000 wei/sec.

someSuperToken.setFlowPermissions(
    alice,
    true,
    false,
    false,
    100000
)
```


Previous
deleteFlow

Next
setMaxFlowPermissions

setMaxFlowPermissions

Function Header

```
/**
 * @dev Update permissions for flow operator - give operator max permissions
 * @param token The token used in flow
 * @param flowOperator The address given flow permissions
 */
function setMaxFlowPermissions(
    ISuperToken token,
    address flowOperator
) internal returns (bool)
```

Example Usage

```
// Giving Alice total permission to manage flows of someSuperToken from the contract

someSuperToken.setFlowPermissions(
    alice
```

```
    address flowOperator
) internal returns (bool)
```

Example Usage

```
// Giving Alice total permission to manage flows of someSuperToken from the contract

someSuperToken.setFlowPermissions(
    alice
)
```

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

Previous
setFlowPermissions

Next
revokeFlowPermissions

Last modified 1mo ago

revokeFlowPermissions

Function Header

```
/**
 * @dev Update permissions for flow operator - revoke all permission
 * @param token The token used in flow
 * @param flowOperator The address given flow permissions
 */
function revokeFlowPermissions(
    ISuperToken token,
    address flowOperator
) internal returns (bool)
```

Example Usage

```
// Removing all of Alice's permissions to manage flows from the contract

someSuperToken.revokeFlowPermissions(
    alice
```

```
    address flowOperator
) internal returns (bool)
```

Example Usage

```
// Removing all of Alice's permissions to manage flows from the contract

someSuperToken.revokeFlowPermissions(
    alice
)
```

super-examples/projects/tradeable-cashflow/contracts/RedirectAll.sol at 0b90a34602...
GitHub

Previous
setMaxFlowPermissions

Next
createFlowFrom

Last modified 1mo ago

createFlowFrom

Creating a flow on behalf of a certain sender account using ACL permissions

Function Header

```
/**
 * @dev Creates flow as an operator without userData
 * @param token The token to flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function createFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate
) internal returns (bool)
```

Example Usage

```
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function createFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate
) internal returns (bool)
```

Example Usage

super-examples/projects/money-streaming-intro/money-streaming-intro-hardhat/co...
GitHub

Previous
revokeFlowPermissions

Next
updateFlowFrom

Last modified 1mo ago

updateFlowFrom

Update a flow on behalf of a certain sender account using ACL permissions

Function Header

```
/**
 * @dev Updates flow as an operator without userData
 * @param token The token to flow
 * @param sender The sender of the flow
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function updateFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate
) internal returns (bool)
```

Example Usage

```
 * @param receiver The receiver of the flow
 * @param flowRate The desired flowRate
 */
function updateFlowFrom(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate
) internal returns (bool)
```

Example Usage

super-examples/projects/money-streaming-intro/money-streaming-intro-hardhat/co...
GitHub

← Previous
createFlowFrom

Next
deleteFlowFrom →

Last modified 1mo ago

deleteFlowFrom

Delete a flow on behalf of a certain sender account using ACL permissions

Function Header

```
/**
* @dev Deletes flow as an operator without userData
* @param token The token to flow
* @param sender The sender of the flow
* @param receiver The receiver of the flow
*/
function deleteFlowFrom(
   ISuperToken token,
   address sender,
   address receiver
) internal returns (bool)
```

Example Usage

```
* @param token The token to flow
* @param sender The sender of the flow
* @param receiver The receiver of the flow
*/
function deleteFlowFrom(
   ISuperToken token,
   address sender,
   address receiver
) internal returns (bool)
```

Example Usage

super-examples/projects/money-streaming-intro/money-streaming-intro-hardhat/co...
GitHub

Previous
updateFlowFrom

Next
With User Data

Last modified 1mo ago

With User Data

Superfluid offers you the option to provide additional data you'd like to pass along with your function call via the userData parameter which is overloaded onto the create/update/delete-Flow functions.

```
// Same function call just with additional parameter for user data
token.createFlow(address receiver, int96 flowRate, bytes memory userData);
token.updateFlow(address receiver, int96 flowRate, bytes memory userData);
token.deleteFlow(address sender, address receiver, bytes memory userData);
```

You can learn more about the usefulness of user data here.

← Previous
deleteFlowFrom

Next
With Context →

Last modified 1mo ago

With User Data

Superfluid offers you the option to provide additional data you'd like to pass along with your function call via the userData parameter which is overloaded onto the create/update/delete-Flow functions.

```
// Same function call just with additional parameter for user data
token.createFlow(address receiver, int96 flowRate, bytes memory userData);
token.updateFlow(address receiver, int96 flowRate, bytes memory userData);
token.deleteFlow(address sender, address receiver, bytes memory userData);
```

You can learn more about the usefulness of user data here.

← Previous
deleteFlowFrom

Next
With Context →

Last modified 1mo ago

With Context

As you can learn about here, Super Agreement calls (like stream creations) in Super App callbacks require the updating of a context bytes variable. That context is returned at the end of the callback.

Below, the `newCtx` is the context bytes variable that will be updated with each Super Agreement call.

```
// Example Super App Callback

function onFlowCreated(
    ISuperToken superToken,
    address sender,
    bytes calldata ctx
) internal override returns (bytes memory /*newCtx*/) {

     newCtx = ctx; // `newCtx` is context bytes variable for updating

     // ... callback logic

}
```

So, to do CFA operations inside of Super App callbacks, you'll need to use the **-WithCtx** versions of each function. These calls all return the updated context (a bytes memory)

```
// We're assuming here that newCtx is what you've named the context bytes
// object that will be updated throughout the callback and returned

// Without user data

token.createFlowWithCtx(
    address receiver,
    int96 flowRate,
    bytes memory ctx // Pass in the context bytes variable for updating here
) returns (bytes memory);

// these functions look very similar for updates and deletions

token.updateFlowWithCtx(address receiver, int96 flowRate, bytes memory ctx) returns (by
token.deleteFlowWithCtx(address sender, address receiver, bytes memory ctx) returns (by

// ACL functions also need to be called with context if called in a callback

function createFlowFromWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate,
    bytes memory ctx
```

```
) internal returns (bytes memory newCtx)

function updateFlowFromWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    int96 flowRate,
    bytes memory ctx
) internal returns (bytes memory newCtx)

function deleteFlowWithCtx(
    ISuperToken token,
    address sender,
    address receiver,
    bytes memory ctx
) internal returns (bytes memory newCtx)

function setFlowPermissionsWithCtx(
    ISuperToken token,
    address flowOperator,
    bool allowCreate,
    bool allowUpdate,
    bool allowDelete,
    int96 flowRateAllowance,
    bytes memory ctx
) internal returns (bytes memory newCtx)
function setMaxFlowPermissionsWithCtx(
```

```
    ISuperToken token,
    address flowOperator,
    bytes memory ctx
) internal returns (bytes memory newCtx)
function revokeFlowPermissionsWithCtx(
    ISuperToken token,
    address flowOperator,
    bytes memory ctx
) internal returns (bytes memory newCtx)
```

Example - Here's the callback snippet continued showing the proper syntax

```
// Example Super App Callback
function afterAgreementCreated(
    ISuperToken superToken,
    address agreementClass,
    bytes32, // _agreementId,
    bytes calldata /_agreementData/,
    bytes calldata ,// _cbdata,
    bytes calldata ctx
) external returns (bytes memory newCtx) {

     newCtx = ctx; // `newCtx` is context bytes variable for updating

     // start a stream to another address
```

```
    bytes32, // _agreementId,
    bytes calldata /_agreementData/,
    bytes calldata ,// _cbdata,
    bytes calldata ctx
) external returns (bytes memory newCtx) {

    newCtx = ctx; // `newCtx` is context bytes variable for updating

    // start a stream to another address
    //note that `token` is the asset you want to stream
    newCtx = token.createFlowWithCtx(
        [someReceiverAddress],
        [flow rate],
        newCtx    // notice `newCtx` being passed in and updated here
    );

}
```


Previous
With User Data

Next
SDK Core

SDK Core

Stream money with the Constant Flow Agreement using the SDK Core

The `ConstantFlowAgreementV1` helper class provides access to create/update/delete flows. You can access this via the `Framework` class (`sf.cfaV1`) or initialize this as a standalone class.

Accessing the CFAv1 Contract

Using SDK Core Framework Object

```
// refresher on initializing the Framework
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const sf = await Framework.create({
  chainId: 137,
  provider
});

// access the cfaV1 object via the Framework class
// see below for a complete example
```

```
const flowInfo = await sf.cfaV1.getFlowInfo(...)
```

Direct Initialization

```
import { ConstantFlowAgreementV1 } from "@superfluid-finance/sdk-core";

const config = {
  hostAddress: "0x3E14dC1b13c488a8d5D310918780c983bD5982E7",
  cfaV1Address: "0x6EeE6060f715257b970700bc2656De21dEdF074C",
  idaV1Address: "0xB0aABBA4B2783A72C52956CDEF62d438ecA2d7a1"
};

const cfaV1 = new ConstantFlowAgreementV1({ options: config });
//super tokens can be loaded directly as well
const daix = await sf.loadSuperToken("DAIx");
```

Methods

Read Methods

`getFlow`: Get data on a stream between two accounts

`getNetFlow`: Get the net flow rate of an account for a certain Super Token

`getAccountFlowInfo`: Get summed flow data of an account for a certain Super Token

Write Methods

Create, Update, Delete

`createFlow`: Starts a stream from a sender to a chosen receiver

`updateFlow`: Updates an existing stream from a sender to a chosen receiver

`deleteFlow`: Cancels an existing stream from a sender to a chosen receiver

Create, Update, Delete with ACL Permissions

`createFlowByOperator`: Starts a stream between two accounts using ACL permissions

`updateFlowByOperator`: Updates a stream between two accounts using ACL permissions

`deleteFlowByOperator`: Deletes a stream between two accounts using ACL permissions

Manage ACL Permissions

`updateFlowOperatorPermissions`: Lets a sender set the ACL permissions of another account

`revokeFlowOperatorPermissions`: Revokes all ACL permissions that an account has over the sender's account

Example Usage

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const sf = await Framework.create({
  chainId: 137,
  provider
});

const daix = await sf.loadSuperToken("DAIx");

// Read example
const flowInfo = await daix.getFlow({
  sender: "0x...",
  receiver: "0x...",
  providerOrSigner: provider
});
console.log("flowInfo", flowInfo);

// Write operation example
```

```
  sender: "0x...",
  receiver: "0x...",
  providerOrSigner: provider
});
console.log("flowInfo", flowInfo);

// Write operation example
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const createFlowOperation = daix.createFlow({
  sender: "0x...",
  receiver: "0x...",
  flowRate: "1000000000"
});
const txnResponse = await createFlowOperation.exec(signer);
const txnReceipt = await txnResponse.wait();
// Transaction Complete when code reaches here
```


← Previous
With Context

Next
Read Methods →

Read Methods

Method Catalog

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

await daix.getFlow({
  sender: string,
  receiver: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await daix.getNetFlow({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await daix.getAccountFlowInfo({
  account: string,
```

```
const daix = await sf.loadSuperToken("DAIX");

await daix.getFlow({
  sender: string,
  receiver: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await daix.getNetFlow({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await daix.getAccountFlowInfo({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

← Previous
SDK Core

Next
getFlow →

Last modified 7mo ago

Super Tokens - Solidity

Interacting with Super Tokens in Solidity

Super Tokens have basic ERC20 functionality, some ERC777 functionality, and several functions to accommodate Superfluid Protocol functionality.

Super Token Contract

protocol-monorepo/SuperToken.sol at dev · superfluid-finance/protocol-monorepo
GitHub

Super Token Interface

protocol-monorepo/ISuperToken.sol at dev · superfluid-finance/protocol-monorepo
GitHub

Remix Example Contract

Remix - Ethereum IDE

An example contract that you can deploy to Goerli and use to experiment with Super Tokens in Solidity

Notice that the Super Token interface doesn't appear to show any Super Agreement functionality (i.e. there's no function in them that starts a stream or does a distribution). Why? Basically, Super Agreements are separate contracts! Computing your Super Token balance is a matter of netting the effects of each Super Agreement into your actual balance. This all happens when you view your balance with `balanceOf`. See this explained in our Super Token In-Depth Explainer,

Getting Set To Interact With Super Tokens

Import the Super Token interface in your .sol file.

```
import { ISuperToken } from "@superfluid-finance/ethereum-contracts/contracts/interfaces
```

ERC20 Functions

Super Tokens offer you all the basic ERC20 features. Learn more about these functions here on OpenZeppelin.

Interface

```
 ) external;
function selfBurn(
    address account,
    uint256 amount,
    bytes memory userData
) external;
function selfTransferFrom(
     address sender,
     address spender,
     address recipient,
     uint256 amount
) external;
function selfApproveFor(
     address account,
     address spender,
     uint256 amount
) external;

/**************************************************************************
 * SuperToken extra functions
 **************************************************************************/

function transferAll(address recipient) external;

/**************************************************************************
* ERC20 wrapping
**************************************************************************/
```

```
    function getUnderlyingToken() external view returns(address tokenAddr);
    function upgrade(uint256 amount) external;
    function upgradeTo(address to, uint256 amount, bytes calldata data) external;
    function downgrade(uint256 amount) external;

    /**************************************************************************
    * Batch Operations
    **************************************************************************/

    function operationApprove(
        address account,
        address spender,
        uint256 amount
    ) external;
    function operationTransferFrom(
        address account,
        address spender,
        address recipient,
        uint256 amount
    ) external;
    function operationUpgrade(address account, uint256 amount) external;
    function operationDowngrade(address account, uint256 amount) external;

}
```

```
    function operationApprove(
        address account,
        address spender,
        uint256 amount
    ) external;
    function operationTransferFrom(
        address account,
        address spender,
        address recipient,
        uint256 amount
    ) external;
    function operationUpgrade(address account, uint256 amount) external;
    function operationDowngrade(address account, uint256 amount) external;

}
```


Developers - Previous
Super Tokens

Next
Super Tokens - SDK Core

```
function totalSupply() external view override(IERC777, IERC20) returns (uint256);
function balanceOf(address account) external view override(IERC777, IERC20) returns(uin
function transfer(address recipient, uint256 amount) external override(IERC20) returns
function allowance(address owner, address spender) external override(IERC20) view retur
function approve(address spender, uint256 amount) external override(IERC20) returns (bo
function transferFrom(address sender, address recipient, uint256 amount) external overr
function increaseAllowance(address spender, uint256 addedValue) external returns (bool)
function decreaseAllowance(address spender, uint256 subtractedValue) external returns (
```

Example: Calling `transfer`

```
ISuperToken(superTokenAddress).transfer(receiverAddress, amountToTransfer)
```

ERC20 Wrapping Functions

These are functions for Wrapper Super Tokens to implement such that you can wrap an underlying ERC20 token into and out of the Wrapper Super Token contract. See this quick explainer on Wrapper Super Tokens.

Interface

```
function getUnderlyingToken() external view returns(address tokenAddr);
```

```
function upgrade(uint256 amount) external;
function upgradeTo(address to, uint256 amount, bytes calldata data) external;
function downgrade(uint256 amount) external;
```

Example: Using `upgrade` to wrap tokens into Super Tokens

The `upgrade` function does the wrapping using `transferFrom` to pull the underlying ERC20 token from your wallet. As a result, before you can wrap your tokens into Super Tokens, you need to `approve` the Super Token contract to be able to spend your underlying ERC20.

```
// approving
IERC20(underlyingTokenAddress).approve(superTokenAddress, amountToWrap)

// wrapping
ISuperToken(superTokenAddress).upgrade(amountToWrap);
```

Example: Calling `upgradeTo`

Usage of `upgradeTo` is rare but useful. It allows you to designate another address that will receive the upgraded tokens and call the `tokensReceived` implementation on it. Essentially, it combines the ERC777 `send` with plain `upgrade`.

```
// approving
IERC20(underlyingTokenAddress).approve(superTokenAddress, amountToWrap)
```

```
// using upgradeTo (use "0x" for calldata if you don't want to pass in anything)
ISuperToken(superTokenAddress).upgradeTo(receiverAddress, amountToWrap, "0x");
```

Example: Using `downgrade` to unwrap Super Tokens

Unwrap your Super Tokens into the underlying ERC20 tokens.

```
// unwrapping
ISuperToken(superTokenAddress).downgrade(amountToUnwrap)
```

ERC777 Functions

Super Tokens also have some ERC777 features. Learn more about these functions here on OpenZeppelin. Note that not all ERC777 functionality is available - read more below 👇

ERC777 in Super Tokens

Interface

```
function granularity() external view override(IERC777) returns (uint256);
```

```
function granularity() external view override(IERC777) returns (uint256);
function send(address recipient, uint256 amount, bytes calldata data) external override
function burn(uint256 amount, bytes calldata data) external override(IERC777);
function isOperatorFor(address operator, address tokenHolder) external override(IERC777
function authorizeOperator(address operator) external override(IERC777);
function revokeOperator(address operator) external override(IERC777);
function defaultOperators() external override(IERC777) view returns (address[] memory);
function operatorSend(
    address sender,
    address recipient,
    uint256 amount,
    bytes calldata data,
    bytes calldata operatorData
) external override(IERC777);
function operatorBurn(
    address account,
    uint256 amount,
    bytes calldata data,
    bytes calldata operatorData
) external override(IERC777);
```

Native Asset Functions

These are functions that only Native Asset Super Tokens possess to allow wrapping and unwrapping of native assets (like MATIC or AVAX). While the interface is named ISETH.sol, it is used for all Native

Assets Super Tokens (MATICx, AVAXx, etc.).

Import the ISETH Interface in Your .sol File.

```
import "@superfluid-finance/ethereum-contracts/contracts/interfaces/tokens/ISETH.sol";
```

Interface

```
function upgradeByETH() external payable;
function upgradeByETHTo(address to) external payable;
function downgradeToETH(uint wad) external;
```

Example - Wrapping a Native Asset

```
ISETH(superTokenAddress).{ value : amountToUpgrade }.upgradeByETH()
```

> You can also wrap a native asset by simply transferring it to the appropriate Native Asset Super Token contract where a `receive()` function is implemented to handle the wrapping on receipt.

Full Super Token Interface

See our GitHub for full comments

protocol-monorepo/ISuperToken.sol at dev · superfluid-finance/protocol-monorepo
GitHub

```
interface ISuperToken is ISuperfluidToken, TokenInfo, IERC20, IERC777 {

    function initialize(
        IERC20 underlyingToken,
        uint8 underlyingDecimals,
        string calldata n,
        string calldata s
    ) external;

    /**************************************************************************
    * TokenInfo & ERC777
    **************************************************************************/

    function name() external view override(IERC777, TokenInfo) returns (string memory);
    function symbol() external view override(IERC777, TokenInfo) returns (string memory
    function decimals() external view override(TokenInfo) returns (uint8);
```

```
/*****************************************************************************
* ERC20 & ERC777
*****************************************************************************/

function totalSupply() external view override(IERC777, IERC20) returns (uint256);
function balanceOf(address account) external view override(IERC777, IERC20) returns


/*****************************************************************************
* ERC20
*****************************************************************************/

function transfer(address recipient, uint256 amount) external override(IERC20) retu
function allowance(address owner, address spender) external override(IERC20) view r
function approve(address spender, uint256 amount) external override(IERC20) returns
function transferFrom(address sender, address recipient, uint256 amount) external o
function increaseAllowance(address spender, uint256 addedValue) external returns (b
function decreaseAllowance(address spender, uint256 subtractedValue) external retur

/*****************************************************************************
* ERC777
*****************************************************************************/

function granularity() external view override(IERC777) returns (uint256);
function send(address recipient, uint256 amount, bytes calldata data) external over
function burn(uint256 amount, bytes calldata data) external override(IERC777);
function isOperatorFor(address operator, address tokenHolder) external override(IER
```

```
function burn(uint256 amount, bytes calldata data) external override(IERC777);
function isOperatorFor(address operator, address tokenHolder) external override(IER
function authorizeOperator(address operator) external override(IERC777);
function revokeOperator(address operator) external override(IERC777);
function defaultOperators() external override(IERC777) view returns (address[] memo
function operatorSend(
    address sender,
    address recipient,
    uint256 amount,
    bytes calldata data,
    bytes calldata operatorData
) external override(IERC777);
function operatorBurn(
    address account,
    uint256 amount,
    bytes calldata data,
    bytes calldata operatorData
) external override(IERC777);

/**************************************************************************
 * SuperToken custom token functions
 **************************************************************************/

function selfMint(
    address account,
    uint256 amount,
    bytes memory userData
```

getFlow

Get information regarding a stream going from a sender to a receiver for a specified Super Token

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let res = await daix.getFlow({
  sender: string,
  receiver: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

Parameters

Parameter	Type	Description
`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver

`providerOrSigner`	`ethers.providers.Provider` *or* `ethers.Signer`	ethers provider or ethers signer object

Return Value(s)

Returns a JSON object literal with the below info:

Key	Value Type	Value Description
`timestamp`	`Date`	Timestamp when the stream of `superToken` was last updated by the account
`flowRate`	`string`	Flow rate (wei/second) of the stream. Always zero or positive.
`deposit`	`string`	Buffer deposit held for the stream
`owedDeposit`	`string`	Additional deposit if the stream is to a Super App

Example Usage

		account
`flowRate`	`string`	Flow rate (wei/second) of the stream. Always zero or positive.
`deposit`	`string`	Buffer deposit held for the stream
`owedDeposit`	`string`	Additional deposit if the stream is to a Super App

Example Usage

super-examples/EmploymentLoan.test.js at 311e5a95d3505dc8c5d8afd420e9edc8b5a...
GitHub

← Previous
Read Methods

Next
getNetFlow →

Last modified 7mo ago

getNetFlow

Get the net flow rate value for an account

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let res = await daix.getNetFlow({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

Parameters

Parameter	Type	Description
`account`	`string`	Address of account you're getting net flow rate for
	`ethers.providers.Pro`	

`providerOrSigner`	`vider` *or* `ethers.Signer`	ethers provider or ethers signer object

Return Value(s)

Returns a string with the below info:

Type	Description
`string`	Net flow rate (wei/second) of `account` for the specified `superToken`. Can be positive or negative as it's the sum of the account's inbound and outbound CFA flow rates.

Example Usage

super-examples/EmploymentLoan.test.js at 311e5a95d3505dc8c5d8afd420e9edc8b5a...
GitHub

Previous
getFlow

Next
getAccountFlowInfo

Returns a string with the below info:

Type	Description
`string`	Net flow rate (wei/second) of `account` for the specified `superToken`. Can be positive or negative as it's the sum of the account's inbound and outbound CFA flow rates.

Example Usage

super-examples/EmploymentLoan.test.js at 311e5a95d3505dc8c5d8afd420e9edc8b5a...
GitHub

← Previous
getFlow

Next
getAccountFlowInfo →

Last modified 7mo ago

getAccountFlowInfo

Get *net* stream information for an account pertaining to a specified Super Token

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let res = await daix.getAccountFlowInfo({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

Parameters

Parameter	Type	Description
`account`	`string`	Address of account you're getting stream information for
	`ethers.providers.Pro`	

`providerOrSigner`	`vider` *or* `ethers.Signer`	ethers provider or ethers signer object

Return Value(s)

Returns a JSON object literal with the below info:

Key	Value Type	Description
`timestamp`	`Date`	Timestamp when a stream of `superToken` was last updated by the account
`flowRate`	`string`	Net flow rate (wei/second) of `account` for the specified `superToken`. Can be positive or negative as it's the sum of the account's inbound and outbound CFA flow rates.
`deposit`	`string`	Total buffer deposits of all streams being sent by `account`
`owedDeposit`	`string`	Total additional deposits for all streams to Super Apps

Key	Value Type	Description
`timestamp`	`Date`	Timestamp when a stream of `superToken` was last updated by the account
`flowRate`	`string`	Net flow rate (wei/second) of `account` for the specified `superToken`. Can be positive or negative as it's the sum of the account's inbound and outbound CFA flow rates.
`deposit`	`string`	Total buffer deposits of all streams being sent by `account`
`owedDeposit`	`string`	Total additional deposits for all streams to Super Apps

← Previous
getNetFlow

Next
Write Methods →

Write Methods

Using SDK Core Write Operations

Write operations are instantiated and then executed with the `exec` function. See below 👇

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

// 1) Instantiate operation to create a 100,000 wei/second stream
let createFlowOp = daix.createFlow({
  sender: "0xAbC...",
  receiver: "0xXyZ...",
  flowRate: "100000",
});

// 2) Execute operation on aliceSigner
await createFlowOp.exec(aliceSigner);
```

Conveniently, this lets you recycle the operation. Say you want to call the same `createFlow`

operation again - you can reuse the operation object on `aliceSigner`!

```
// 3) USe the same operation object to create 100,000 wei/second stream again
await createFlowOp.exec(aliceSigner);
```

Method Catalog

```
// Regular Usage

//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

daix.createFlow({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});

daix.updateFlow({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
```

```
});

daix.deleteFlow({
  sender: string,
  receiver: string,
  userData?: string
});

//ACL Usage

daix.updateFlowOperatorPermissions({
  flowOperator: string,
  permissions: number, // should enter 1-7
  flowRateAllowance: string
});

daix.revokeFlowOperatorPermissions({
  flowOperator: string
})

daix.createFlowByOperator({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});
```

```
  receiver: string,
  flowRate: string,
  userData?: string
});

daix.updateFlowByOperator({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});

daix.deleteFlowByOperator({
  sender: string,
  receiver: string,
  userData?: string
})
```


Previous
getAccountFlowInfo

Next
createFlow

createFlow

Lets a sender create a stream to a receiver.

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.createFlow({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});

await flowOp.exec( <<ethers signer>> ); // should have same address as `sender`
```

Parameters

Parameter	Type	Description

`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`flowRate`	`string`	Desired wei/second flow rate at which the `sender` wants to start streaming to the `receiver`
`userData`	`string`	*Optional* user data

Example Usage

super-examples/EmploymentLoan.test.js at main · superfluid-finance/super-examples
GitHub

Previous
Write Methods

Next
updateFlow

Last modified 7mo ago

`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`flowRate`	`string`	Desired wei/second flow rate at which the `sender` wants to start streaming to the `receiver`
`userData`	`string`	*Optional* user data

Example Usage

super-examples/EmploymentLoan.test.js at main · superfluid-finance/super-examples
GitHub

Previous
Write Methods

Next
updateFlow

Last modified 7mo ago

updateFlow

Lets a sender update an existing stream to a receiver to a new flow rate

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.updateFlow({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});

await flowOp.exec( <<ethers signer>> ); // should have same address as `sender`
```

Parameters

Parameter	Type	Description

`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`flowRate`	`string`	New desired wei/second flow rate at which the `sender` wants to be streaming to the `receiver`
`userData`	`string`	*Optional* user data

Example Usage

super-examples/EmploymentLoan.test.js at main · superfluid-finance/super-examples
GitHub

Additional Considerations

- If `updateFlow` is called with a `flowRate` that is the same as the current flow rate, it will *not* revert.

Previous
createFlow

Next
deleteFlow

		streaming to the `receiver`
`userData`	`string`	*Optional* user data

Example Usage

super-examples/EmploymentLoan.test.js at main · superfluid-finance/super-examples
GitHub

Additional Considerations

- If `updateFlow` is called with a `flowRate` that is the same as the current flow rate, it will *not* revert.

← Previous
createFlow

Next
deleteFlow →

Last modified 7mo ago

deleteFlow

Lets a sender delete an existing stream to a receiver

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.deleteFlow({
  sender: string,
  receiver: string,
  userData?: string
});

await flowOp.exec( <<ethers signer>> ); // should have same address as sender
```

Parameters

Parameter	Type	Description
`sender`	`string`	Address of sender

`receiver`	`string`	Address of receiver
`userData`	`string`	*Optional* user data

Example Usage

super-examples/EmploymentLoan.test.js at main · superfluid-finance/super-examples
GitHub

Additional Considerations

- If there is already no stream from sender to receiver, then `deleteFlow` will revert.

Previous
updateFlow

Next
createFlowByOperator

Last modified 7mo ago

`receiver`	`string`	Address of receiver
`userData`	`string`	*Optional* user data

Example Usage

super-examples/EmploymentLoan.test.js at main · superfluid-finance/super-examples
GitHub

Additional Considerations

- If there is already no stream from sender to receiver, then `deleteFlow` will revert.

Previous
updateFlow

Next
createFlowByOperator

Last modified 7mo ago

createFlowByOperator

Starts a stream from a `sender` to a `receiver` using ACL permissions.

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.createFlowByOperator({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});

await flowOp.exec( <<ethers signer>> );
```

Parameters

Parameter	Type	Description

Parameter	Type	Description
`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`flowRate`	`string`	Desired wei/second flow rate to be created from `sender` to `receiver`
`userData`	`string`	*Optional* user data

Example Usage

https://codesandbox.io/s/create-flow-as-operator-nv607c?from-embed=&file=/src/CreateFlowAsOperator.js:787-851

← Previous
deleteFlow

Next
updateFlowByOperator →

Last modified 7mo ago

Parameter	Type	Description
`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`flowRate`	`string`	Desired wei/second flow rate to be created from `sender` to `receiver`
`userData`	`string`	*Optional* user data

Example Usage

https://codesandbox.io/s/create-flow-as-operator-nv607c?from-embed=&file=/src/CreateFlowAsOperator.js:787-851

← Previous
deleteFlow

Next
updateFlowByOperator →

updateFlowByOperator

Updates an existing stream from a `sender` to a `receiver` using ACL permissions.

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.updateFlowByOperator({
  sender: string,
  receiver: string,
  flowRate: string,
  userData?: string
});

await flowOp.exec( <<ethers signer>> );
```

Parameters

Parameter	Type	Description

`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`flowRate`	`string`	New desired wei/second flow rate that `sender` wants to start streaming at to the `receiver`
`userData`	`string`	*Optional* user data

Example Usage

https://codesandbox.io/s/update-flow-as-operator-xceyyz?from-embed=&file=/src/UpdateFlowAsOperator.js:935-999

Additional Considerations

- If `updateFlowByOperator` is called with a `flowRate` that is the same as the current flow rate, it will *not* revert.

← Previous
createFlowByOperator

Next
deleteFlowByOperator →

`flowRate`	`string`	New desired wei/second flow rate that `sender` wants to start streaming at to the `receiver`
`userData`	`string`	*Optional* user data

Example Usage

https://codesandbox.io/s/update-flow-as-operator-xceyyz?from-embed=&file=/src/UpdateFlowAsOperator.js:935-999

Additional Considerations

- If `updateFlowByOperator` is called with a `flowRate` that is the same as the current flow rate, it will *not* revert.

← Previous
createFlowByOperator

Next
deleteFlowByOperator →

deleteFlowByOperator

Deletes an existing stream from a `sender` to a `receiver` using ACL permissions.

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.deleteFlowByOperator({
  sender: string,
  receiver: string,
  userData?: string
});

await flowOp.exec( <<ethers signer>> );
```

Parameters

Parameter	Type	Description
`sender`	`string`	Address of sender

`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`userData`	`string`	*Optional* user data

Example Usage

https://codesandbox.io/s/delete-flow-as-operator-rfk6pd?from-embed=&file=/src/DeleteFlowAsOperator.js

Additional Considerations

- If there is already no stream from sender to receiver, then `deleteFlowByOperator` will revert.

← Previous
updateFlowByOperator

Next
updateFlowOperatorPermissions →

Last modified 7mo ago

Parameter	Type	Description
`sender`	`string`	Address of sender
`receiver`	`string`	Address of receiver
`userData`	`string`	*Optional* user data

Example Usage

https://codesandbox.io/s/delete-flow-as-operator-rfk6pd?from-embed=&file=/src/DeleteFlowAsOperator.js

Additional Considerations

- If there is already no stream from sender to receiver, then `deleteFlowByOperator` will revert.

← Previous
updateFlowByOperator

Next
updateFlowOperatorPermissions →

Super Tokens - SDK Core

Perform basic transfers, approvals, token wrapping & unwrapping using the SDK Core

The `SuperToken` class can also be accessed via the `Framework` class and allows you read from/write to the blockchain. It also provides write functions for both the CFAV1 and IDAV1 contracts in the context of the token. That is, the token field for these different methods will be the token address specified during the creation of this class.

Framework based initialization

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const sf = await Framework.create({
  networkName: "matic",
  provider
```

```
});

const usdcx = await sf.loadSuperToken("0xCAa7349CEA390F89641fe306D93591f87595dc1F");
```

Direct Initialization

```
import { SuperToken } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const config = {
  hostAddress: "0x3E14dC1b13c488a8d5D310918780c983bD5982E7",
  cfaV1Address: "0x6EeE6060f715257b970700bc2656De21dEdF074C",
  idaV1Address: "0xB0aABBA4B2783A72C52956CDEF62d438ecA2d7a1"
};

const usdcx = await SuperToken.create({
  address: "0xCAa7349CEA390F89641fe306D93591f87595dc1F",
  config,
  networkName: "matic", // you can also pass in chainId instead (e.g. chainId: 137)
  provider
});
```

SuperToken Functions

```
const usdcx = await sf.loadSuperToken("0xCAa7349CEA390F89641fe306D93591f87595dc1F");

// ERC20 `Token`
// Read functions
await usdcx.balanceOf({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.allowance({
  owner: string,
  spender: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.name({
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.symbol({
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

```
await usdcx.totalSupply({
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});


// Write operations
usdcx.approve({
  receiver: string,
  amount: string
});

usdcx.transfer({
  receiver: string,
  amount: string
});

usdcx.transferFrom({
  sender: string,
  receiver: string,
  amount: string
});

// `SuperToken` only function
await usdcx.realtimeBalanceOf({
  account: string,
  timestamp: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

```
});

// Write Functions

// All write functions return Promise<Operation>

// SuperToken Write operations
usdcx.downgrade({ amount: string });

usdcx.upgrade({ amount: string });

// SuperToken CFAV1/IDAV1 Functions are the same as the
// ConstantFlowAgreementV1/InstantDistributionAgreementV1 class functions
// except instead of the sf.cfaV1/idaV1.function() signature, it is token.function()
// e.g. await usdcx.createIndex({ indexId: "0", userData: "0x" }).exec(signer);
// and you don't need to pass in a token as a parameter as it uses the token address
// of the instantiated class.
```

Example Usage

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);
```

```
const sf = await Framework.create({
  networkName: "matic",
  provider
});

const usdcx = await sf.loadSuperToken("0xCAa7349CEA390F89641fe306D93591f87595dc1F");

// Read example
const name = await usdcx.name();
console.log(name);

// Write operation example
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const transferOperation = usdcx.transfer({ receiver: "0x...", amount: "1000000" });
const txnResponse = await transferOperation.exec(signer);
const txnReceipt = await txnResponse.wait();
// Transaction Complete when code reaches here
```

> Note: you can also get the underlying Token object which only has ERC20 token read/write methods-this is useful for things like approving token spend to a SuperToken contract prior to upgrading for example.

```
const usdc = usdcx.underlyingToken;
```

```
// Write operation example
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const transferOperation = usdcx.transfer({ receiver: "0x...", amount: "1000000" });
const txnResponse = await transferOperation.exec(signer);
const txnReceipt = await txnResponse.wait();
// Transaction Complete when code reaches here
```

> Note: you can also get the underlying Token object which only has ERC20 token read/write methods-this is useful for things like approving token spend to a SuperToken contract prior to upgrading for example.

```
const usdc = usdcx.underlyingToken;
const totalSupply = await usdc.totalSupply();
```


← Previous
Super Tokens - Solidity

Next
Super Tokens - Frontend Examples →

updateFlowOperatorPermissions

Lets the caller grant another account the desired ACL permissions for streams of a certain Super Token.

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.updateFlowOperatorPermissions({
  flowOperator: string,
  permissions: number,
  flowRateAllowance: string
});

await flowOp.exec( <<ethers signer>> );
```

Parameters

Parameter	Type	Description

`flowOperator`	`string`	Address of the account whose ACL permissions the caller is updating
`permissions`	`number`	Number indicating level of permission to be granted to the account.
`flowRateAllowance`	`string`	Maximum amount of flow rate increase that the flowOperator is permitted - learn more Akin to allowance on ERC20

Example Usage

https://codesandbox.io/s/updateflowpermissions-example-3k8hor?from-embed=&file=/src/UpdateFlowPermissions.js:810-891

← Previous deleteFlowByOperator

Next revokeFlowOperatorPermissions →

		updating
`permissions`	`number`	Number indicating level of permission to be granted to the account.
`flowRateAllowance`	`string`	Maximum amount of flow rate increase that the flowOperator is permitted - learn more Akin to allowance on ERC20

Example Usage

https://codesandbox.io/s/updateflowpermissions-example-3k8hor?from-embed=&file=/src/UpdateFlowPermissions.js:810-891

← Previous
deleteFlowByOperator

Next
revokeFlowOperatorPermissions →

Last modified 7mo ago

revokeFlowOperatorPermissions

Revokes all ACL permissions that an account has over the caller's account for streams of a certain Super Token.

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

let flowOp = daix.revokeFlowOperatorPermissions({
  flowOperator: string
});

await flowOp.exec( <<ethers signer>> );
```

Parameters

Parameter	Type	Description
`flowOperator`	`string`	Address of the account whose ACL permissions the caller is revoking

```
});

await flowOp.exec( <<ethers signer>> );
```

Parameters

Parameter	Type	Description
`flowOperator`	`string`	Address of the account whose ACL permissions the caller is revoking

Example Usage

https://codesandbox.io/s/revoke-flow-permissions-example-ednt8t?from-embed=&file=/src/RevokeFlowPermissions.js:767-851

← Previous
updateFlowOperatorPermissions

Next
Frontend Examples →

Last modified 7mo ago

Frontend Examples

Getting Started with the Constant Flow Agreement

Let's walk through creating Superfluid streams using the Superfluid Core-SDK. We'll get straight into the code with some examples, but you can scroll down to learn about how the SDK works in more detail.

> NOTE: we recommend using the Superfluid Console and the Superfluid Dashboard as you go through these interactive tutorials. They'll help you view your streams in action.

Creating a Flow

Creating a flow requires you to call `superToken.createFlow()` and pass in the receiver and flowRate required for the flow. There's also an optional field you can use called `userData` - which you can learn more about in our guide on the topic.

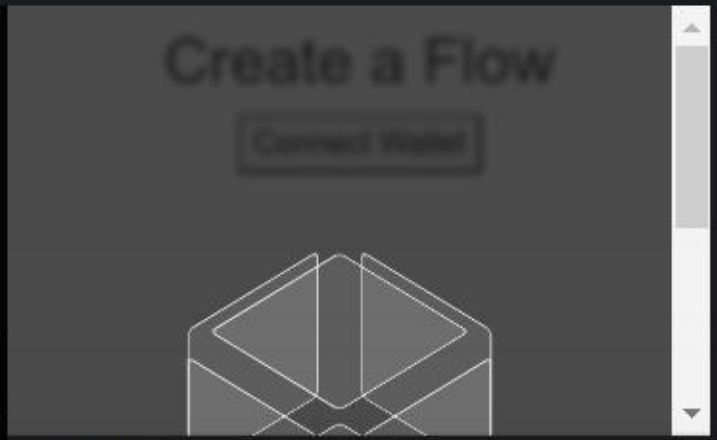
See below for an example of how this works. You can enter your own Ethereum address and `flowRate` (i.e. how much DAIx you want to send in wei/second) to send yourself a stream of DAIx.

Scroll down in the React App page (on the right side of the screen below) to see your flowRate converted into DAIx per month.

> NOTE: these examples require test ether/MATIC and test super tokens. You can get some from our faucet here.

CreateFlow.js

```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
4    Button,
5    Form,
6    FormGroup,
7    FormControl,
8    Spinner,
9    Card
10 } from "react-bootstrap";
11 import "./createFlow.css";
12 import { ethers } from "ethers";
13
14 let account;
15
```

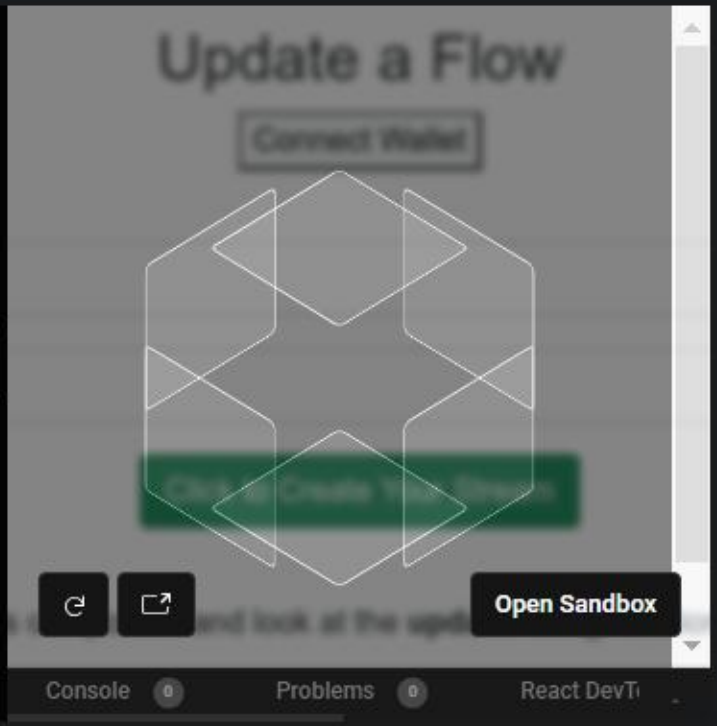


Updating a Flow

Updating a flow is very similar to creating one. You'll need to pass the same parameters that you passed in the create flow function to `superToken.updateFlow()` - except this time you'll need to pass in a new `flowRate`. If you attempt to edit a stream which does not currently exist, this function will revert. Give it a try:

App.js

```
import "./styles.css";

import { UpdateFlow } from "./Upd

export default function App() {
  return (
    <div className="App">
      <UpdateFlow />
    </div>
  );
}

```

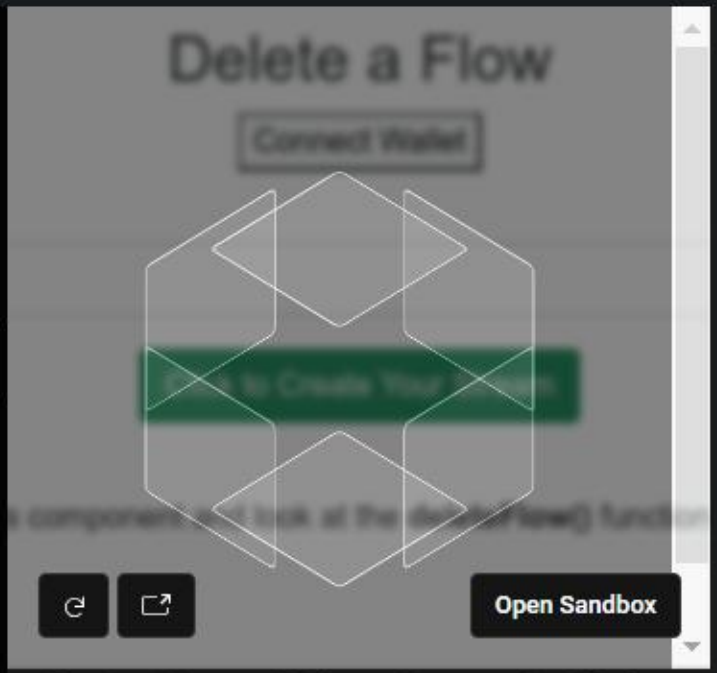


Deleting a Flow

Finally, we can delete flows. To delete a flow, you'll just need to pass the `sender`, `receiver` to the `superToken.deleteFlow()` function. If you attempt to delete a flow which does not exist, it will revert.

DeleteFlow.js

```
import React, { useState, useEffe
import { Framework } from "@super
import {
  Button,
  Form,
  FormGroup,
  FormControl,
  Spinner,
  Card
} from "react-bootstrap";
import "./updateFlow.css";
import { ethers } from "ethers";

let account;
```



Money Streaming

A money stream (or 'flow' - you'll see these terms used interchangeably) is a continuous stream of

funds sent from one account to another. You can open up a stream by passing a two arguments to the `superToken.createFlow()` function within our SDK:

`flowRate` - the amount of tokens being streamed per second, denominated in wei (i.e. with 18 decimals)

`receiver` - the recipient of the money stream

`superToken` - the token being streamed

Once you call this function, the protocol will take a small deposit up front in escrow. On testnets, this amount is 1 hour worth of the stream, and on mainnets, it is 4 hours worth of the stream.

> **Note**: if you send funds to a 'super app' contract, these deposit amounts will double. You can learn more about this in our Super App guide.

Once the stream is opened, it will stay open as long as the `sender` account has a balance > 0 of the `superToken` being used for the stream. If you let your balance hit zero, your deposit will be liquidated by the network, and your stream will be closed.

In the above examples, we've created a small React App which makes use of the Superfluid Core-SDK. You will see an example of creating, updating, and deleting flows. Each interactive section can be opened and forked for your own projects.

You'll see that we're importing the Superfluid SDK in each key component, then initializing the SDK using the following logic:

```
// initialization of the core SDK

const sf = await Framework.create({
   chainId: Number,
   provider: ethersProvider
});
```

In our case, we're using a link from Alchemy to get our own custom HTTP provider for the Kovan network. You could easily change this so that you use a `web3Provider` (i.e. a metamask connection in the browser) instead of an RPC URL. You can see the SDK's README for an example of this.

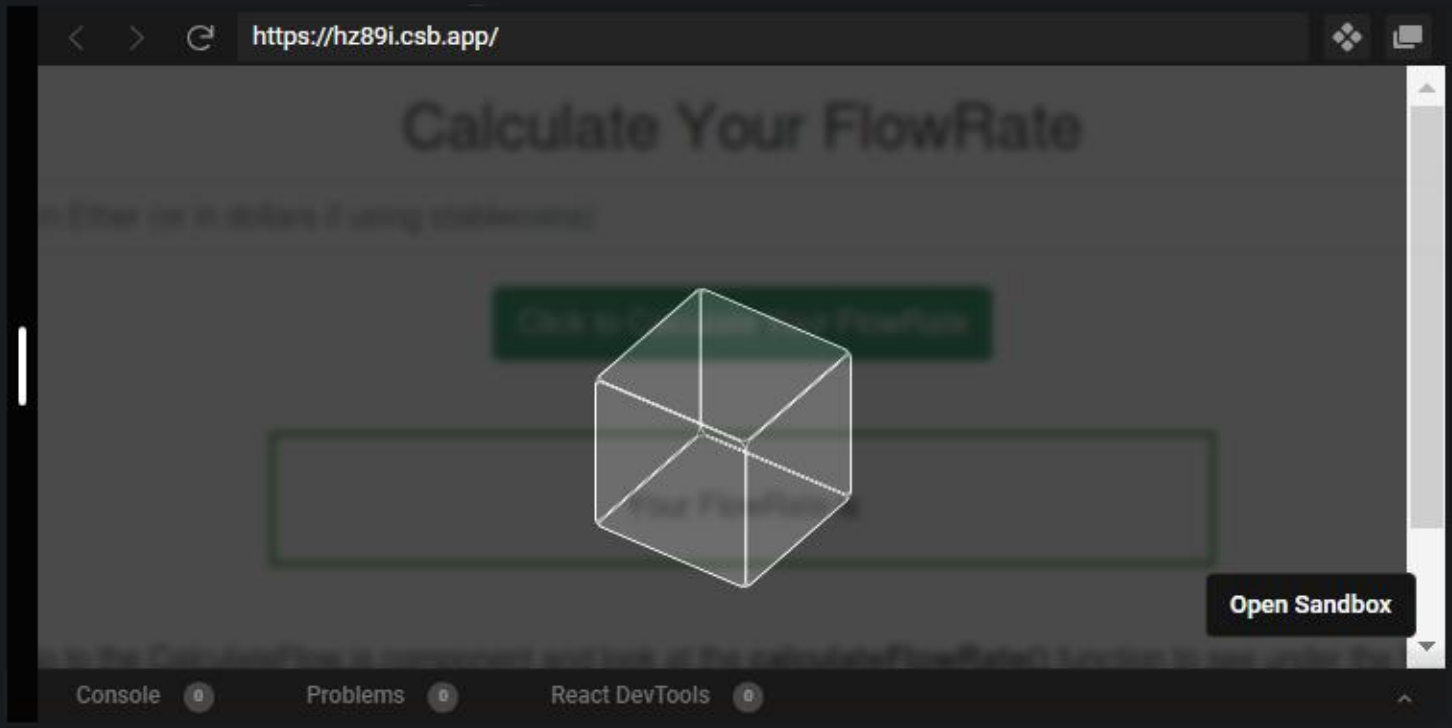
We've also provided an unlocked address that is loaded up with test net Super Tokens for example purposes. We've used the `sf.createSigner()` method to create a signer with this address.

```
// Creating a signer
  const signer = sf.createSigner({
    privateKey:
      "0x...", //private key goes here
    provider: customHttpProvider
  });
```

FlowRates

The flowRate is the # of tokens you'd like to send per second, denominated in wei. You can use the

The flowRate is the # of tokens you'd like to send per second, denominated in wei. You can use the calculator below to get the flowRate for an amount of tokens you want to send per month. You can pass this calculated flowRate into the examples above to create & update your streams.



Next, we'll follow a similar process for working with the Instant Distribution Agreement.

← Previous
revokeFlowOperatorPermissions

Next
Access Control List (ACL) →

Next, we'll follow a similar process for working with the Instant Distribution Agreement.

← Previous
revokeFlowOperatorPermissions

Next
Access Control List (ACL) →

Last modified 1mo ago

WAS THIS PAGE HELPFUL?

Access Control List (ACL)

An approval layer for creating, updating, or deleting streams on another user's behalf.

The Access Control List (ACL) allows any account (plain EOA or contract) to configure permissions for another account to create/update/delete streams on its behalf.

It's basically the money streaming equivalent of the ERC20 approval mechanism!

	ERC20	Money Streams
View account permissions	`allowance()`	`getFlowOperatorData`
Grant permissions	`allowance()`	`updateFlowOperatorPermissions`

moving tokens with permissions	**`transferFrom()`**	`(create/update/delete)`**`FlowByOperator`**

`allowance`, `approval`, and `transferFrom` are to ERC20 transfers as `getFlowOperatorData`, `updateFlowOperatorPermissions`, and `(create/update/delete)FlowByOperator` are to Superfluid streams.

Before checking out the functions ACL provides you, you'll want to understand the `flowRateAllowance` and `permissions` parameters.

> NOTE: this page is a deep dive into how the access control list system works at the protocol level. We recommend using the SuperTokenV1Library and the functions found here to give allowances to flowOperators & to create, update, and delete flows as an operator.

`flowRateAllowance` Parameter

You can think of `flowRateAllowance` as a tank. When you set `flowRateAllowance` in your `updateFlowOperatorPermissions` call imagine you've filled up that tank to a level of your choosing for the `flowOperator` account.

That `flowOperator` account can spend its `flowRateAllowance` tank as it likes. Every time it **increases** a flow rate on your behalf, its `flowRateAllowance` tank is depleted by the amount of

increases a flow rate on your behalf, its `flowRateAllowance` tank is depleted by the amount of the increase.

On the other side, actions that **decrease** your flow rate (updating a stream to a lower flow rate or deleting one) **don't affect** `flowRateAllowance`.

If a `flowOperator`'s `flowRateAllowance` is too small for it to create/increase a flow, then its action is reverted. This entire dynamic is just like the ERC20 allowance mechanism! Here's an example to drive it home 👇

Example:

Let's assume that we enable an operator to create and update streams with a flowRate allowance of 1000 tokens per month (which would be represented as a flow rate of `385802469135802` tokens per second). If our operator creates a stream on our behalf with a `flowRate` of 500 tokens per month, then the operator's remaining `flowRateAllowance` becomes 500 tokens per month (`1000/month - 500/month`). Then, if our operator updates another one of our streams that is already outstanding by increasing our `flowRate` by 250 tokens per month, then our operator's `flowRateAllowance` will decrease by another 250 tokens per month, leaving the remaining flowRateAllowance at 250 tokens per month (`500/month - 250/month`).

`permissions` Parameter

The `permissions` value is a `uint256` from 1 to 7 representing a specific level of access that an operator gets over your streams. You pass in `permissions` when calling `updateFlowOperatorPermissions`.

Permission Type	Parameter Value
Create	1
Update	2
Create or Update	3
Delete	4
Create or Delete	5
Delete or Update	6
Create, Update, or Delete	7

Granting Permissions

`updateFlowOperatorPermissions`

Set permissions.

```
/**
   * @dev msgSender from `ctx` updates permissions for the `flowOperator` with `flowRate
   * @param token Super token address
```

```
    * @param flowOperator The permission grantee address
    * @param permissions A bitmask representation of the granted permissions
    * @param flowRateAllowance The flow rate allowance the `flowOperator` is granted (on
    * @param ctx Context bytes (see ISuperfluid.sol for Context struct)
  */
  function updateFlowOperatorPermissions(
          ISuperfluidToken token,
          address flowOperator,
          uint8 permissions,
          int96 flowRateAllowance,
          bytes calldata ctx
      )
          external virtual
          returns(bytes memory newCtx);
```

`authorizeFlowOperatorWithFullControl`

Conveniently grant **full** operator permissions to an account. That account is now an operator with full control will not have their `flowRateAllowance` deducted when it hike the flow rates of the account it has control over.

```
  /**
      * @dev msgSender from `ctx` grants `flowOperator` create/update/delete permissions
      * @param token Super token address
```

```
 * @param flowOperator The permission grantee address
 * @param ctx Context bytes (see ISuperfluid.sol for Context struct)
 */
function authorizeFlowOperatorWithFullControl(
    ISuperfluidToken token,
    address flowOperator,
    bytes calldata ctx
)
    external virtual
    returns(bytes memory newCtx);
```

`revokeFlowOperatorWithFullControl`

Conveniently **revoke all** operator permissions from an account.

```
/**
 * @notice msgSender from `ctx` revokes `flowOperator` create/update/delete permiss
 * @dev `permissions` and `flowRateAllowance` will both be set to 0
 * @param token Super token address
 * @param flowOperator The permission grantee address
 * @param ctx Context bytes (see ISuperfluid.sol for Context struct)
 */
function revokeFlowOperatorWithFullControl(
    ISuperfluidToken token,
    address flowOperator,
```

```
        bytes calldata ctx
    )
        external virtual
        returns(bytes memory newCtx);
```

Create, Update, Delete Streams with Operator Controls

When it comes to actually creating, updating, or deleting a stream by an operator, you can use functions with the `ByOperator` suffix to perform these operations.

`createFlowByOperator`

Creates a stream between two accounts using operator controls.

- Reduces `flowRateAllowance` given to operator over `sender`
- Reverts if `permissions` and `flowRateAllowance` are insufficient over `sender`

```
/**
    * @notice Create a flow between sender and receiver
    * @dev A flow created by an approved flow operator (see above for details on callba
    * @param token Super token address
    * @param sender Flow sender address (has granted permissions)
    * @param receiver Flow receiver address
```

```
    * @param flowRate New flow rate in amount per second
    * @param ctx Context bytes (see ISuperfluid.sol for Context struct)
    */
    function createFlowByOperator(
        ISuperfluidToken token,
        address sender,
        address receiver,
        int96 flowRate,
        bytes calldata ctx
    )
        external virtual
        returns(bytes memory newCtx);
```

`updateFlowByOperator`

Updates an existing stream's rate using operator controls. Reduces `flowRateAllowance` given to operator if it is for an increase.

- If it's an increase (update to a higher rate), reduces `flowRateAllowance` given to operator over `sender` by `flowRate - (old flow rate)`
- Reverts if `permissions` and `flowRateAllowance` are insufficient over `sender`

```
/**
    * @notice Update a flow between sender and receiver
```

```
    * @dev A flow updated by an approved flow operator (see above for details on callbac
    * @param token Super token address
    * @param sender Flow sender address (has granted permissions)
    * @param receiver Flow receiver address
    * @param flowRate New flow rate in amount per second
    * @param ctx Context bytes (see ISuperfluid.sol for Context struct)
    */
    function updateFlowByOperator(
        ISuperfluidToken token,
        address sender,
        address receiver,
        int96 flowRate,
        bytes calldata ctx
    )
        external virtual
        returns(bytes memory newCtx);
```

`deleteFlowByOperator`

Deletes a stream between two accounts using operator controls.

```
/**
     * @notice Delete the flow between sender and receiver
     * @dev A flow deleted by an approved flow operator (see above for details on callba
```

```
/**
     * @notice Delete the flow between sender and receiver
     * @dev A flow deleted by an approved flow operator (see above for details on callba
     * @param token Super token address
     * @param ctx Context bytes (see ISuperfluid.sol for Context struct)
     * @param receiver Flow receiver address
     */
    function deleteFlowByOperator(
        ISuperfluidToken token,
        address sender,
        address receiver,
        bytes calldata ctx
    )
        external virtual
        returns(bytes memory newCtx);
```


Previous
Frontend Examples

Next
ACL Features - SDK

ACL Features - SDK

Giving other accounts control over stream operations

Granting Permissions to Flow Operators

Using the new ACL features, you may grant other accounts the ability to create, update, and/or delete streams on your behalf. You can read more about these features, and how to implement them in solidity in this page:

Access Control List (ACL)

You can also make use of the ACL features using the SDK Core. This page will provide sandbox examples for managing flow permissions & performing CRUD operations for streams on the front end.

NOTE: we recommend using the Superfluid Console and the Superfluid Dashboard as you

go through these interactive tutorials. They'll help you view your streams in action.

Updating & Revoking Flow Permissions

If you'd like to grant full control over streams on a specific token to an operator, you can use the `authorizeFlowOperatorWithFullControl` function and pass in the `flowOperator` and `superToken` you're granting the authorization on. However, in this example we will show you how to grant specific permission levels using the `updateFlowOperatorPermissions` function.

To grant a flow rate allowance to an operator, you'll need to pass in the following to the `updateFlowOperatorPermissions` function:

`operator` - the address of the operator you're giving the allowance to

`permissions` - the permission type you're granting the operator. Note, you will pass in a number from 1-7 to represent the permission type. You can find more information on this here. This table will help you determine the permission type:

Permission Type	Parameter Value
Create	1
Update	2
Create or Update	3

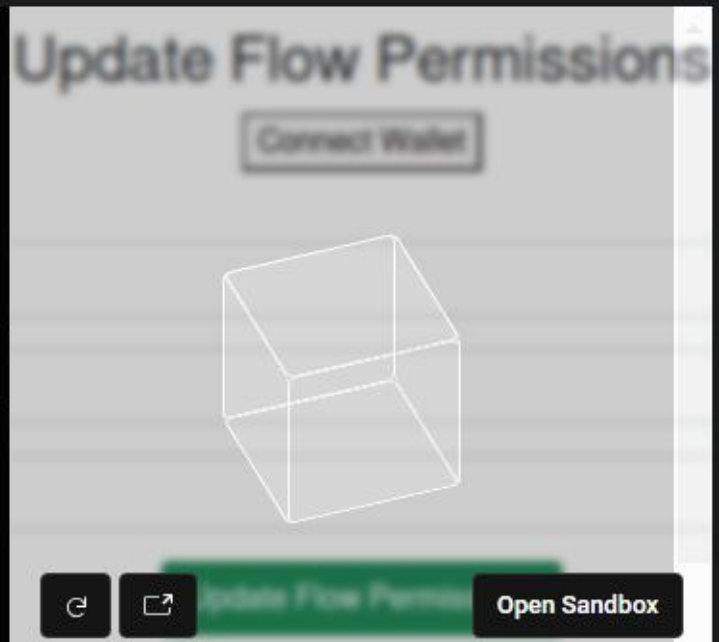
Delete	4
Create or Delete	5
Delete or Update	6
Create, Update, or Delete	7

`flowRateAllowance` - the total allowance you're providing the operator.

`superToken` - the token on which the allowance is being provided

UpdateFlowPermissions.js

```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
4    Button,
5    Form,
6    FormGroup,
7    FormControl,
8    Spinner,
9    Card
10 } from "react-bootstrap";
11 import "./createFlow.css";
12 import { ethers } from "ethers";
13
14 let account;
```



```
let account;
//where the Superfluid logic take
```

Console 4 Problems 0 React DevT

Updating flow permissions

To revoke permissions, you can use the `revokeFlowOperatorWithFullControl` function. To revoke permissions from the flow operator, you'll need to pass in the:

`flowOperator` - the operator with the flowAllowance

`superToken` - the token on which the permissions were granted

App.js

```
import "./styles.css";

import { BatchCall } from "./

export default function App(
  return (
    <div className="App">
      <BatchCall />
    </div>
  );
}

```

https://tdp4g.csb.app/

Batch Calls

Upgrade and create a flow in a single tx 🤯

Connect Wallet

Enter how many tokens to approve first

Click to approve tokens

Enter the dollar amount you'd l

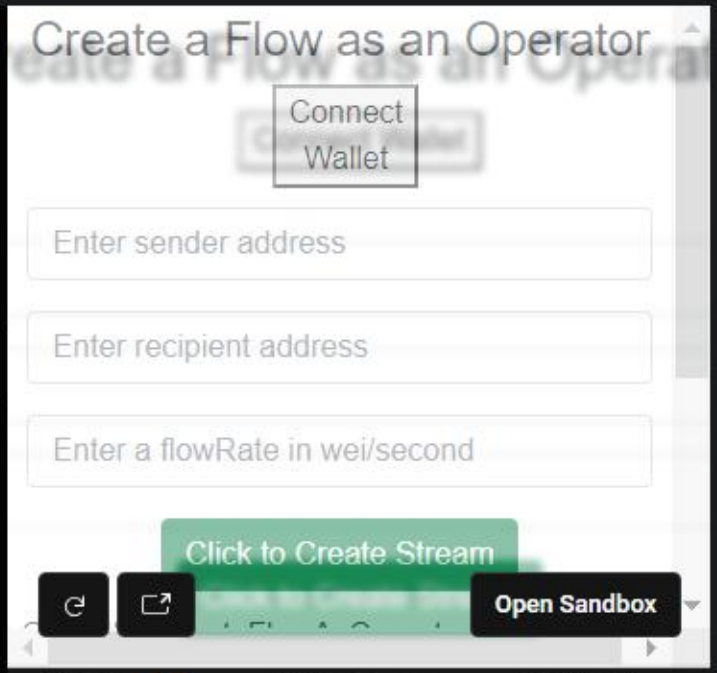
Open Sandbox

From here, creating, updating, and deleting streams will work in a quite straightforward way. You can find examples of each below.

> NOTE: updating a flow with an increased flow rate or creating flows will decrement the operator's flow rate allowance, but updating a flow with a decreased flow rate or deleting a flow will not. You can find more information on how flow rate allowances work here.

CreateFlowAsOperator.js

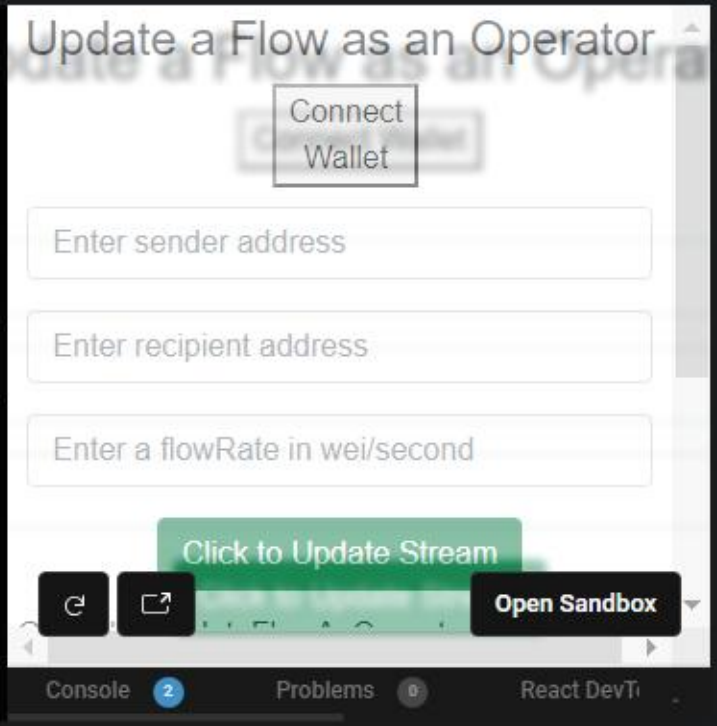
```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
4    Button,
5    Form,
6    FormGroup,
7    FormControl,
8    Spinner,
9    Card
10 } from "react-bootstrap";
11 import "./createFlow.css";
12 import { ethers } from "ethers";
13
14 // let account;
15
```



Creating a flow as an operator

UpdateFlowAsOperator.js

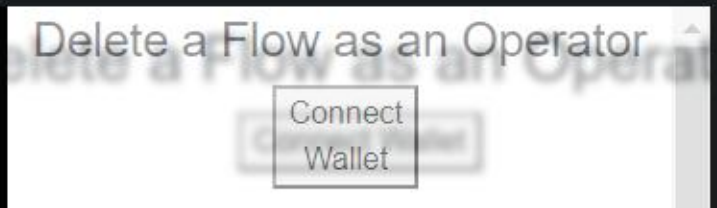
```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
4    Button,
5    Form,
6    FormGroup,
7    FormControl,
8    Spinner,
9    Card
10 } from "react-bootstrap";
11 import "./updateFlow.css";
12 import { ethers } from "ethers";
13
14 // let account;
15
```



Updating a flow as an operator

DeleteFlowAsOperator.js

```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
```



```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
4    Button,
5    Form,
6    FormGroup,
7    FormControl,
8    Spinner,
9    Card
10 } from "react-bootstrap";
11 import "./updateFlow.css";
12 import { ethers } from "ethers";
13
14 // let account;
15
```

Connect Wallet

Enter sender address

Enter recipient address

Click to Delete Stream

Go to the DeleteFlowAsOperator.js component

Open Sandbox

Console 2 Problems 0 React DevT

Deleting a flow as an operator

Previous
Access Control List (ACL)

Next
More...

Last modified 6mo ago

WAS THIS PAGE HELPFUL?

More...

Understand how to properly translate your desired flow rate to wei/sec - the dimension that's needed for flow operations.

Flow Rate Time Frames

Learn about the usage of the Flow NFTs that the Superfluid Protocol releases to the sender and receiver of active streams.

Flow NFTs

A quick guide on executing many streams at once using Safe transaction batching.

Building Batched Streams in Safe

Flow Rate Time Frames

Learn about the usage of the Flow NFTs that the Superfluid Protocol releases to the sender and receiver of active streams.

Flow NFTs

A quick guide on executing many streams at once using Safe transaction batching.

Building Batched Streams in Safe

Previous
ACL Features - SDK

Next
Flow Rate Time Frames

Last modified 2mo ago

Flow Rate Time Frames

Understand flow rate time denominations better

Superfluid Protocol accepts flow rates in wei/second! When you call `createFlow` or do any kind of operation where a `flowRate` parameter is required, be that with our Solidity library or with our SDK, you have to pass in a per-second rate.

So when I say I'm streaming you 10 DAIx/month, what flow rate am I really choosing under the hood? Read on to understand how to properly translate your desired flow rate into wei/second *as per our standards*.

Doing your math differently that what's suggested here may result in differences between how your flow rate shows between your expectations and what we show on our Superfluid Dashboard and Superfluid Console.

Reframing Flow Rates To Wei/Second

From Month

Ex: Convert a flow rate of 10 DAIx/month to a wei/sec flow rate.

```
Monthly Time Frame = 10 DAIx / month
                   = 10 DAIx / (30 * 24 * 60 * 60)
                   = 10 * (10^18) / ( (60 * 60 * 24 * 365) / 12 )
Wei/Sec Flow Rate  = 3805175038052 wei / second
```

> If you didn't follow our recommendation and did this calculation by dividing by `(60 * 60 * 24 * 30)`, you'd end up with a slightly larger wei/sec flow rate of `3805175038052` which would result in your stream showing as slightly larger on our Dashboard and Console.

From Year

Ex: Convert a flow rate of 100 DAIx/year to a wei/sec flow rate.

```
Monthly Time Frame = 100 DAIx / year
                   = 100 DAIx / (60 * 60 * 24 * 365)
                   = 100 * (10^18) / (60 * 60 * 24 * 365)
Wei/Sec Flow Rate  = 3170979198376 wei / second
```

← Previous
More...

Next
Building Batched Streams in Safe →

> If you didn't follow our recommendation and did this calculation by dividing by `(60 * 60 * 24 * 30)`, you'd end up with a slightly larger wei/sec flow rate of `3805175038052` which would result in your stream showing as slightly larger on our Dashboard and Console.

From Year

Ex: Convert a flow rate of 100 DAIx/year to a wei/sec flow rate.

```
Monthly Time Frame = 100 DAIx / year
                   = 100 DAIx / (60 * 60 * 24 * 365)
                   = 100 * (10^18) / (60 * 60 * 24 * 365)
Wei/Sec Flow Rate  = 3170979198376 wei / second
```


← Previous
More...

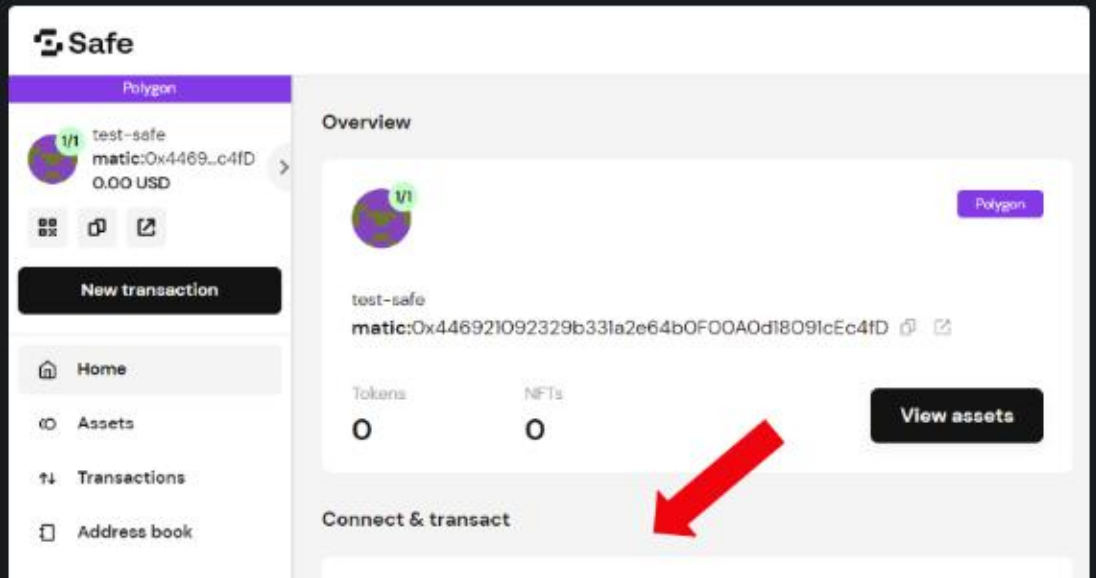
Next
Building Batched Streams in Safe →

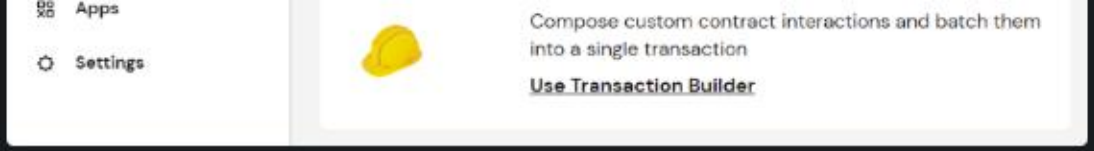
Building Batched Streams in Safe

Send multiple streams in the same transaction with Safe transaction batching

If you have a Safe and are looking to send multiple streams in one transaction rather than having to manually execute each one, this guide will show you how it's done.

1. Go the Safe Transaction Builder





2. Get CFAv1Forwarder Address

Find the CFAv1Forwarder address for the desired network below:

Network	Address
Ethereum Mainnet Polygon Gnosis Chain Optimism Arbitrum Avalanche BNB Chain Goerli Arbitrum Goerli Optimism Goerli Mumbai	0xcfA132E353cB4E398080B9700609bb008eceB125
Avalanche Fuji	0x2CDd45c5182602a36d391F7F16DD9f8386C3bD8D

Paste that address in the *Enter Address or ENS Name* box on the Transaction Builder interface.

3. Copy CFAv1Forwarder ABI

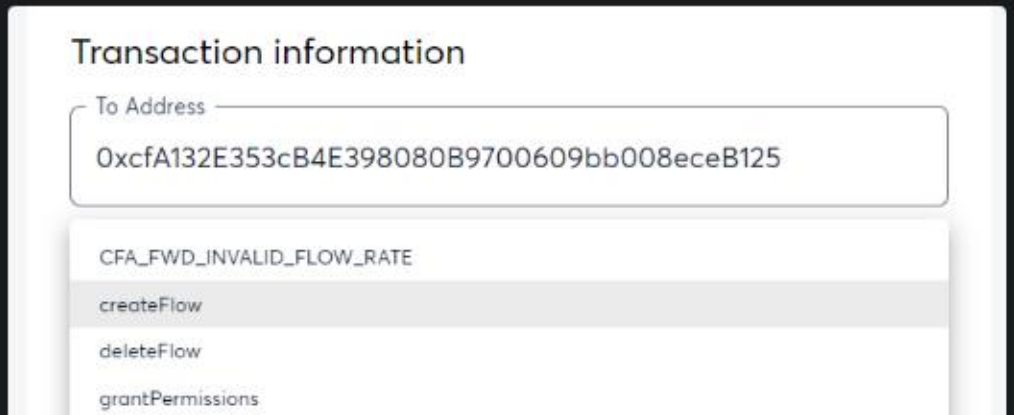
Copy the ABI below and paste in the *Enter ABI* box in the Transaction Builder interface.

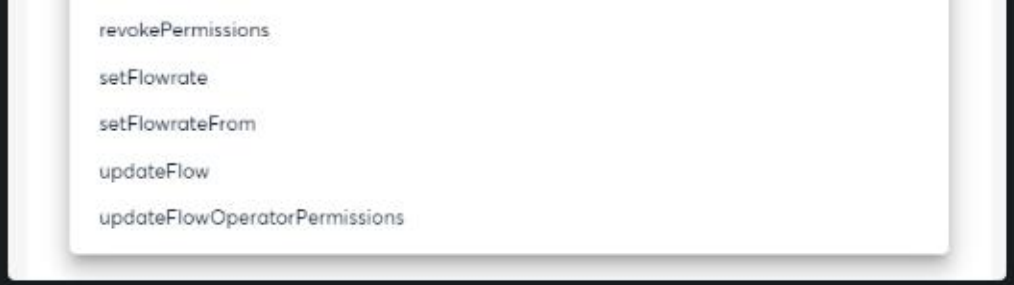
> **Expand for CFAv1Forwarder ABI**

4. Line Up Streaming Transactions

Now you'll be able to set up each stream transaction for the batch one at a time.

If you're looking to create streams, select the `createFlow` method.





Function Parameters

`token (address)` - Super Token you're streaming

`sender (address)` - Your Safe address

`receiver (address)` - The receiver address

`flowrate (int96)` - Flow rate in seconds

> Flow Rate is always in wei/second. Super Tokens always have 18 decimals of precision.
>
> For example: You want to stream **1000 DAIx/month** to Alice
>
> `flowrate` = 1,000 * (10^18) / (seconds in month)
>
> `flowrate` = 1,000 * (10^18) / ((60 * 60 * 24 * 365) / 12)
>
> `flowrate` = 1,000 * (10^18) / 2628000
>
> `flowrate` = 380517503805175

```
flowrate = 1,000 * (10^18) / 2628000
flowrate = 380517503805175
```

`userData (bytes)` - put in `0x`

Hit `Add Transaction` and repeat for each stream you'd like to send 🔁

5. Create and Send the Batch

Once you added all operations to the batch click "Create Batch" and then "Send Batch". After execution, your streams should be running!

You can check out the Superfluid Safe App to see the streams in action and conduct necessary modifications.

← Previous
Flow Rate Time Frames

Next
Flow NFTs →

```
flowrate = 1,000 * (10^18) / 2628000
flowrate = 380517503805175
```

`userData (bytes)` - put in `0x`

Hit `Add Transaction` and repeat for each stream you'd like to send 🔁

5. Create and Send the Batch

Once you added all operations to the batch click "Create Batch" and then "Send Batch". After execution, your streams should be running!

You can check out the Superfluid Safe App to see the streams in action and conduct necessary modifications.

← Previous
Flow Rate Time Frames

Next
Flow NFTs →

Flow NFTs

NFTs that are held by the sender and receiver of an active stream

What are Flow NFTs

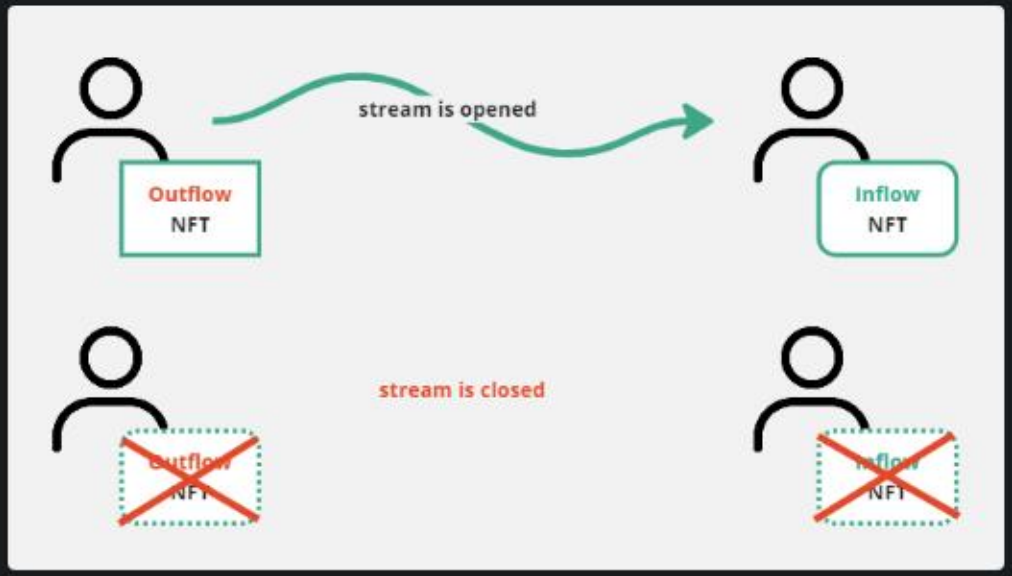
You can think of Flow NFTs as on-chain labels of the sender and receiver of a certain stream. They enhance the visibility of streams on-chain.

Flow NFTs are non-transferrable ERC721 tokens that are minted when a stream is opened and burned when closed.

Purpose of Flow NFTs

1. Provide a visual representation of streams for display - akin to how Uniswap v3 NFTs allow you to view your LP positions on NFT market place galleries.
2. Allow for easy viewing of sender and receiver pertaining to stream creation/deletion transactions via block explorer and through minting/burning events - for example.

Flow NFT Dynamics



Opening a stream

When a stream is started between two accounts, the Superfluid Protocol mints both the sender and receiver an NFT. The sender gets an *Outflow NFT* and the receiver gets an *Inflow NFT*.

Closing a stream

When a stream is cancelled, both the sender and receiver's Flow NFTs are burned.

Transferring Flow NFTs

Flow NFTs are non-transferrable.

> **Each Super Token has its own distinct Inflow and Outflow NFT contracts.**
>
> For instance, if a fUSDCx stream is started to you and then a fDAIx stream is started to you on Mumbai testnet, the Inflow NFTs you receive will originate from separate contracts. The fUSDCx Inflow NFT address is here while the fDAIx address is here. They are not the same contract!

Example Transaction

See the minting of Flow NFTs in action in this create-flow transaction.

Stream Sender: `0xbbf19110bf3c536b608877f1e02fdbd3cfac65a6`

Stream Receiver: `0xc41876dab61de145093b6aa87417326b24ae4ecd`

Polygon Transaction Hash (Txhash) Details | PolygonScan
Polygon (MATIC) Blockchain Explorer

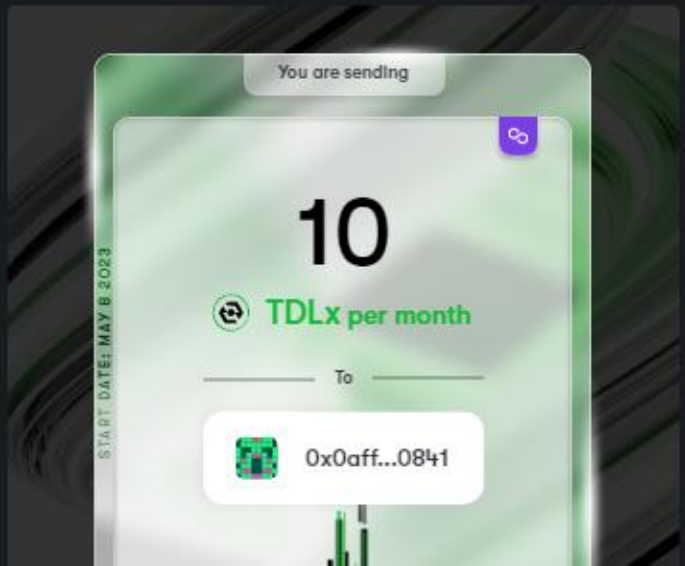
As you can see, through minting of the Flow NFTs, you can view on-chain who is sending and receiving the streams involved in a transaction.

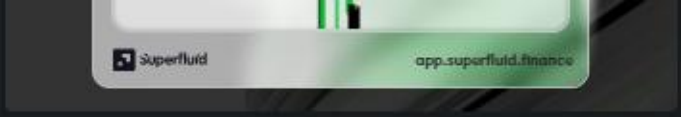
Flow NFT Visuals

The Flow NFT visuals will always show:

1. Monthly Flow Rate
2. Super Token being streamed
3. Sender (for Inflow NFTs) or Receiver (for Outflow NFTs)
4. Network, indicated by a small flag on the top right
5. Start day of stream, shown in small font on left edge

Example **Inflow** NFT for an account being streamed TDLx





Example **Outflow** NFT for an account streaming TDLx

Flow NFT API

Generate Flow NFT SVGs and metadata using our Superfluid NFT API

GitHub - superfluid-finance/sfnft
GitHub

Useful Functions

Super Token Functions

Super Tokens hold two public immutable variables containing the Inflow and Outflow NFT addresses

```
address outflowNFT = superToken.CONSTANT_OUTFLOW_NFT();

address inflowNFT = superToken.CONSTANT_INFLOW_NFT();
```

Flow NFT Functions

Flow NFTs are ERC721s. On top of the ERC721 interface, these read functions they provide allow you to work with the stream data pertaining to the Flow NFTs.

```
struct FlowNFTData {
    address superToken;
    address flowSender;
    address flowReceiver;
    uint32 flowStartDate;
}

// Get stream data from a Flow NFT token ID
function flowDataByTokenId(
    uint256 tokenId
) external view returns (FlowNFTData memory flowData);

// Get Flow NFT token ID given some stream data
function getTokenId(
    address superToken,
    address flowSender,
    address flowReceiver
) external view returns (uint256);
```

```
    address superToken,
    address flowSender;
    address flowReceiver;
    uint32 flowStartDate;
}

// Get stream data from a Flow NFT token ID
function flowDataByTokenId(
    uint256 tokenId
) external view returns (FlowNFTData memory flowData);

// Get Flow NFT token ID given some stream data
function getTokenId(
    address superToken,
    address flowSender,
    address flowReceiver
) external view returns (uint256);
```


Previous
Building Batched Streams in Safe

Next - Developers
Distributions (IDA)

Distributions (IDA)

Scalable one-to-many distributions

What is the IDA?

Recurring one-to-many distributions are hard to scale; the more receivers there are, the more transfers must be done, which runs up gas costs.

The Instant Distribution Agreement (IDA) makes recurring one-to-many distributions scalable. It does so by letting a sending account assign proportions to many receiving accounts and distribute tokens to them on the basis of the set proportions in a single transaction at a fixed gas cost. Here's how it works 👇

1. First, a sender (called a "publisher") publishes an IDA index which will work like a channel organizing how Super Tokens will proportionally distribute to receivers (called "subscribers").

2. Then, the publisher can set proportions for various subscriber accounts under the IDA Index. This is done by issuing units to the IDA Index which work like distribution shares. The more units an account has relative to other receivers, the more it will receive of each distribution.

> **NOTE:** When working within solidity, precision errors can happen when the result of a

division operation is not a whole number. In the case of the IDA, these precision errors rear their head with *very small distribution values* and *really large unit values*. However, keep in mind that there is a tradeoff: if you keep the total amount of units too low, then this prevents you from maintaining a flexible ratio between members of a pool.

In most cases, it's unlikely that your distribution values will be low enough to cause problems (because super tokens have 18 trailing decimals anyway). However, there *is* a chance that you will run into problems with unit values which are too small or too large.

You want units small enough to avoid precision errors while still being accurate. Our recommendation is to use somewhere between 7 and 9 trailing decimals for your IDA unit values. For example, if you have an index that is to be split into thirds, break it up like so:

```
subscriber1 = 100000000

subscriber2 = 100000000

subscriber3 = 100000000

totalUnits = 300000000
```

3. When Super Tokens are distributed through the IDA Index, every subscriber will instantly receive tokens *in proportion* to the amount of units they have over the total outstanding units. This is done all at once with a fixed gas cost.

tokens *in proportion* to the amount of units they have over the total outstanding units. This is done all at once with a fixed gas cost.

> **NOTE**: Instant Distribution Agreements are NOT one-and-done; they do not reset after a distribution is triggered. Each receiver's units for the index will persist across distributions, so the sender can continue to distribute through the index as many times and in various amounts as desired. A receiver's units can be increased or decreased as the sender sees fit.

This section will show you how to develop code that interacts with the IDA. For something more conceptual, check out this simple conceptual breakdown 👇

Distributions (IDA)

← Previous
Flow NFTs

Next
IDA - Solidity →

tokens *in proportion* to the amount of units they have over the total outstanding units. This is done all at once with a fixed gas cost.

> **NOTE**: Instant Distribution Agreements are NOT one-and-done; they do not reset after a distribution is triggered. Each receiver's units for the index will persist across distributions, so the sender can continue to distribute through the index as many times and in various amounts as desired. A receiver's units can be increased or decreased as the sender sees fit.

This section will show you how to develop code that interacts with the IDA. For something more conceptual, check out this simple conceptual breakdown 👇

Distributions (IDA)

← Previous
Flow NFTs

Next
IDA - Solidity →

Super Tokens - Frontend Examples

How to Wrap & Unwrap Existing ERC20 Assets

Wrapping and Unwrapping ERC20 Tokens

You can easily wrap and unwrap ERC20s & native assets as super tokens using the Superfluid dashboard, but you might want to write code to allow your users to do the same in your application. In the below examples, we show you 2 code examples which makes it easy to do this using the Superfluid SDK-Core. We've used an address that is unlocked and has lots of fake DAI (fDAI) and Super fake DAI (fDAIx) tokens. Feel free to play around and use this code for testing!

> ⓘ NOTE: these examples require test ether/MATIC and test super tokens. You can get some from our faucet here.

Super Token Front End Example

Enter an amount to approve and upgrade in the form field below in whole dollar amounts. Make sure you 'Click to Approve Token Upgrade' first, and wait a few seconds after the console shows that the transaction has been completed before you 'Click to Upgrade Your Tokens.'

In this example, we need to `approve` the super token wrapper contract to spend our DAI before we can call `upgrade` .

Next, we can choose to unwrap our ERC20 tokens by calling `downgrade` on the super token contract to burn the corresponding super tokens and receive the ERC20 assets back in return.

> **NOTE**: depending on your device, you may need to scroll down in the browser view below to view & click the button to upgrade tokens.

Previous
Super Tokens - SDK Core

Next
Types of Super Tokens

IDA - Solidity

The Super Token Library allows you to work with instant distributions in Solidity

SuperTokenV1Library Contract

protocol-monorepo/SuperTokenV1Library.sol at dev · superfluid-finance/protocol-mo...
GitHub

Example Code & Guide

super-examples/projects/instant-distribution-intro at main · superfluid-finance/super...
GitHub

Simple contract showing usage of many IDA functions

Agreement Abstraction

The objective of the `SuperTokenV1Library` is to abstract the code required to call an agreement

The objective of the SuperTokenV1Library is to abstract the code required to call an agreement. Below is a comparative example of how an index might be created with and without the library.

```
function createWithoutLibrary() external {
    _host.callAgreement(
        _ida,
        abi.encodeWithSelector(
            _ida.createIndex.selector,
            publisher,
            indexId,
            new bytes(0)
        ),
        new bytes(0)
    );
}

function createWithLibrary(ISuperToken token) external {
    token.createIndex(indexId);
}
```

Importing and Initialization

```
import {
    ISuperfluid,
    ISuperToken
} from "@superfluid-finance/ethereum-contracts/contracts/interfaces/superfluid/ISuperflu
```

```
) internal;

// usage
token.distribute(indexId, amount, userData);
```

Distribute in a Super App Callback

```
// library function declaration
function distributeWithCtx(
    ISuperToken token,
    uint32 indexId,
    uint256 amount,
    bytes memory ctx,
) internal returns (bytes memory newCtx);

// usage
return token.distributeWithCtx(indexId, amount, ctx);
```

Approve Subscription

Approves a subscription to an index. This is called by the subscriber to the index and can be called even before units are issued to the subscriber, though the index must at least exist first.

```
// library function declaration
```

```
// library function declaration
function approveSubscription(
    ISuperToken token,
    address publisher,
    uint32 indexId
) internal;

// usage
token.approveSubscription(publisher, indexId);
```

Approve Subscription with User Data

```
// library function declaration
function approveSubscription(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    bytes memory userData
) internal;

// usage
token.approveSubscription(publisher, indexId, userData);
```

Approve Subscription in a Super App Callback

```
// library function declaration
function approveSubscriptionWithCtx(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    bytes memory ctx,
) internal returns (bytes memory newCtx);

// usage
return token.approveSubscriptionWithCtx(publisher, indexId, ctx);
```

Revoke Subscription

Revokes a previously approved subscription. This is called by the subscriber.

```
// library function declaration
function revokeSubscription(
    ISuperToken token,
    address publisher,
    uint32 indexId
) internal;

// usage
```

```
token.revokeSubscription(publisher, indexId);
```

Revoke Subscription with User Data

```
// library function declaration
function revokeSubscription(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    bytes memory userData
) internal;

// usage
token.revokeSubscription(publisher, indexId, userData);
```

Revoke Subscription in a Super App Callback

```
// library function declaration
function revokeSubscriptionWithCtx(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    bytes memory ctx,
) internal returns (bytes memory newCtx);
```

```
// usage
return token.revokeSubscriptionWithCtx(publisher, indexId, ctx);
```

Update Subscription Units

Updates the number of units, or "shares", of the index assigned to a subscriber. This is called by the publisher of the index.

```
// library function declaration
function updateSubscriptionUnits(
    ISuperToken token,
    uint32 indexId,
    address subscriber,
    uint128 units
) internal;

// usage
token.updateSubscriptionUnits(indexId, subscriber, units);
```

Update Subscription Units with User Data

```
// library function declaration
function updateSubscriptionUnits(
```

```
    ISuperToken token,
    uint32 indexId,
    address subscriber,
    uint128 units,
    bytes memory userData
) internal;

// usage
token.updateSubscriptionUnits(indexId, subscriber, units, userData);
```

Update Subscription Units in a Super App Callback

```
// library function declaration
function updateSubscriptionUnitsWithCtx(
    ISuperToken token,
    uint32 indexId,
    address subscriber,
    uint128 units,
    bytes memory ctx,
) internal returns (bytes memory newCtx);

return token.updateSubscriptionUnitsWithCtx(indexId, subscriber, units, ctx)
```

Delete Subscription

Deletes an existing subscription, setting the subscriber's units to zero. This is called by the publsiher.

```
// library function declaration
function deleteSubscription(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    address subscriber
) internal;

// usage
token.deleteSubscription(publisher, indexId, subscriber);
```

Delete Subscription with User Data

```
// library function declaration
function deleteSubscription(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    address subscriber,
    bytes memory userData
) internal;
```

```
// usage
token.deleteSubscription(publisher, indexId, subscriber, userData);
```

Delete Subscription in a Super App Callback

```
// library function declaration
function deleteSubscriptionWithCtx(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    address subscriber,
    bytes memory ctx,
) internal returns (bytes memory newCtx);

// usage
return token.deleteSubscriptionWithCtx(publisher, indexId, subscriber, ctx);
```

Claim

Claims a pending distribution of an index. This is called by the subscriber and updates their real time balance instantly.

```
// library function declaration
function claim(
```

```
function claim(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    address subscriber
) internal;

// usage
token.claim(publisher, indexId, subscriber);
```

Claim with User Data

```
// library function declaration
function claim(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    address subscriber,
    bytes memory userData
) internal;

// usage
token.claim(publisher, indexId, subscriber, userData);
```

Claim in a Super App Callback

```
token.claim(publisher, indexId, subscriber, userData);
```

Claim in a Super App Callback

```
// library function declaration
function claimWithCtx(
    ISuperToken token,
    address publisher,
    uint32 indexId,
    address subscriber,
    bytes memory ctx,
) internal returns (bytes memory newCtx);

// usage
return token.claimWithCtx(publisher, indexId subscriber, ctx);
```


← Developers - Previous
Distributions (IDA)

Next
IDA - SDK Core →

```
} from "@superfluid-finance/ethereum-contracts/contracts/interfaces/superfluid/ISuperfl

import {
    IInstantDistributionAgreementV1
} from "@superfluid-finance/ethereum-contracts/contracts/interfaces/agreements/IInstantD

import {
    SuperTokenV1Library
} from "@superfluid-finance/ethereum-contracts/contracts/apps/SuperTokenV1Library.sol";

contract MyContract {
    // use the SuperTokenV1Library for the ISuperToken type
    using SuperTokenV1Libary for ISuperToken;

    ISuperToken public token;

    constructor(
        ISuperToken _token,
    ) {

       token = _token
    }

    // your code here...
}
```

Basic Usage

Once this is initialized, we can use the library to create, read, update, and delete IDAv1 agreements as demonstrated in this `createIndex` example.

```
function myFunction(ISuperToken token, uint32 indexId) {
    token.createIndex(indexId);
}
```

Callback Usage

To use this library inside of a Super App callback, you will need to use the equivalent function with `WithCtx` at the end of it. For example, instead of creating an index in a callback with `createIndex`, you should use `createIndexWithCtx` and return the result, `newCtx`, of type `bytes memory`.

```
function afterAgreementCreated(
    ISuperToken superToken,
    address /*agreementClass*/,
    bytes32 /*agreementId*/,
    bytes calldata /*agreementData*/,
    bytes calldata /*cbdata*/,
    bytes calldata ctx
} external override returns (bytes memory newCtx) {
```

```
    require(msg.sender == address(host), "only host");
    uint32 indexId = 0;
    return superToken.createIndexWithCtx(indexId, ctx);

}
```

API Specification

The following documents the library function's declaration along with the usage the function, assuming the `InitData` struct is named `_idav1Lib`.

Each function has four variants:

1. First is the standard function shown in the Basic Usage section above.
2. Second is an override of the standard usage to include arbitrary user data.
3. Third is the `WithCtx` function shown in the Callback Usage section above.
4. Fourth is an override of the `WithCtx` function to include arbitrary user data.

Note that each function below includes the "library function declaration". This is not necessary to write in your own code and is simply there for reference. The code following "usage" is what you would write in your own contract.

For more information on each parameter, please refer to the code comment documentation for each function in the library.

Create Index

Creates an index with a super token and an index id. The function caller is the publisher of the index.

```
// library function declaration
function createIndex(
    ISuperToken token,
    uint32 indexId
) internal;

// usage
token.createIndex(indexId);
```

Create Index with User Data

```
// library function declaration
function createIndex(
    ISuperToken token,
    uint32 indexId,
    bytes memory userData
) internal;

// usage
```

```
token.createIndex(indexId, userData);
```

Create Index in a Super App Callback

```
// library function declaration
function createIndexWithCtx(
    ISuperToken token,
    uint32 indexId,
    bytes memory ctx // ctx passed to the callback function
) internal returns (bytes memory newCtx);

// usage
return token.createIndexWithCtx(indexId, ctx);
```

Update Index Value

Updates the value of the index. This updates the real time balances of all approved subscribers in a single transaction. Notice that this is similar to `distribute`, except here you must specify the new total index value in the `indexValue` parameter. This fails if it is not greater than the last index value.

```
// library function decalaration
function updateIndexValue(
    ISuperToken token,
```

```
    ISuperToken token,
    uint32 indexId,
    uint128 indexValue
) internal;

// usage
token.updateIndexValue(indexId, indexValue);
```

Update Index Value with User Data

```
// library function declaration
function updateIndexValue(
    ISuperToken token,
    uint32 indexId,
    uint128 indexValue,
    bytes memory userData
) internal;

// usage
token.updateIndexvalue(indexId, indexValue, userData);
```

Update Index Value in a Super App Callback

```
// library function declaration
function updateIndexValueWithCtx(
    ISuperToken token,
    uint32 indexId,
    uint128 indexValue,
    bytes memory ctx,
) internal returns(bytes memory newCtx);

// usage
return token.updateIndexValueWithCtx(indexId, indexValue, ctx);
```

Update Index Value in a Super App Callback

```
// library function declaration
function updateIndexValueWithCtx(
    ISuperToken token,
    uint32 indexId,
    uint128 indexValue,
    bytes memory ctx,
) internal returns(bytes memory newCtx);

// usage
return token.updateIndexValueWithCtx(indexId, indexValue, ctx);
```

Distribute

This function is functionally similar to `updateIndexValue`, but instead of having to specify the new `indexValue`, you can pass an `amount` by which the `indexValue` should be incremented. This is simply another way to distribute tokens.

```
// library function declaration
function distribute(
    ISuperToken token,
    uint32 indexId,
    uint256 amount
) internal;

// usage
token.distribute(indexId, amount);
```

Distribute with User Data

```
// library function declaration
function distribute(
    ISuperToken token,
    uint32 indexId,
    uint256 amount,
    bytes memory userData
```

IDA - SDK Core

Send one to many transactions with the Instant Distribution Agreement using the SDK Core

The `InstantDistributionAgreementV1` helper class provides access to a variety of IDA functions. You can access this via the `Framework` class (`sf.idaV1`) or initialize this as a standalone class.

Using the `Framework` Class

```
// Some code
import { Framework } from "@superfluid.finance/sdk-core"
import { ethers } from "ethers";

const sf = await Framework.create({
  chainId: 137,
  provider
});

//access the idaV1 object via the Framework class
//see below for a complete example
```

```
const flowInfo = await sf.idaV1.getSubscription(...)
```

Direct Initialization

```
import { InstantDistributionAgreementV1 } from "@superfluid-finance/sdk-core";

const config = {
  hostAddress: "0x3E14dC1b13c488a8d5D310918780c983bD5982E7",
  cfaV1Address: "0x6EeE6060f715257b970700bc2656De21dEdF074C",
  idaV1Address: "0xB0aABBA4B2783A72C52956CDEF62d438ecA2d7a1"
};

//load a super token - this can be done by symbol or address
const daix = await sf.loadSuperToken("DAIx");
const idaV1 = new InstantDistributionAgreementV1({ options: config });
```

IDAV1 Functions

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

// Read functions
await daix.getSubscription({
```

```
  publisher: string,
  indexId: string,
  subscriber: string,
  providerOrSigner: string
});

await daix.getIndex({
  publisher: string,
  indexId: string,
  providerOrSigner: string
});


// Write operations
daix.createIndex({
  indexId: string,
  userData?: string
});

daix.distribute({
  indexId: string,
  amount: string,
  userData?: string
});

daix.updateIndexValue({
  indexId: string,
```

```
  indexValue: string,
  userData?: string
});

daix.updateSubscriptionUnits({
  indexId: string,
  subscriber: string,
  units: string,
  userData?: string
});

daix.approveSubscription({
  indexId: string,
  publisher: string,
  userData?: string
});

daix.revokeSubscription({
  indexId: string,
  publisher: string,
  userData?: string
});

daix.deleteSubscription({
  indexId: string,
  subscriber: string,
  publisher: string,
```

```
  userData?: string
});

daix.claim({
  indexId: string,
  subscriber: string,
  publisher: string,
  userData?: string
});
```

Example Usage

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const sf = await Framework.create({
  chainId: 137,
  provider
});

//load the token you'd like to use like this
```

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

// Read example
const subscription = await daix.getSubscription({ publisher: "0x...", indexId: "1", sub
console.log(subscription);


// Write operation example
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const createIndexOperation = daix.createIndex({ indexId: "0", userData: "0x" });
const txnResponse = await createIndexOperation.exec(signer);
const txnReceipt = await txnResponse.wait();
// Transaction Complete when code reaches here
```


Previous
IDA - Solidity

Next
IDA - Frontend Examples

```
//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
const daix = await sf.loadSuperToken("DAIx");

// Read example
const subscription = await daix.getSubscription({ publisher: "0x...", indexId: "1", sub
console.log(subscription);


// Write operation example
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const createIndexOperation = daix.createIndex({ indexId: "0", userData: "0x" });
const txnResponse = await createIndexOperation.exec(signer);
const txnReceipt = await txnResponse.wait();
// Transaction Complete when code reaches here
```


Previous
IDA - Solidity

Next
IDA - Frontend Examples

IDA - Frontend Examples

Create an index and distribute tokens using the Superfluid Core-SDK

Introduction

Let's take a look at another Superfluid Agreement - the **Instant Distribution Agreement (IDA).** This agreement can be used to make one-time-payments to multiple recipients. The IDA is great way to implement a revenue sharing system or issue airdrops. It consists of a **Publishing Index** with `indexId`, an `indexValue`, and one or more **subscribers** who receive varying amounts of `units` within the Index.

> NOTE: we recommend using the Superfluid Console and the Superfluid Dashboard as you go through these interactive tutorials. They'll help you view your IDA interactions.

Create an Index

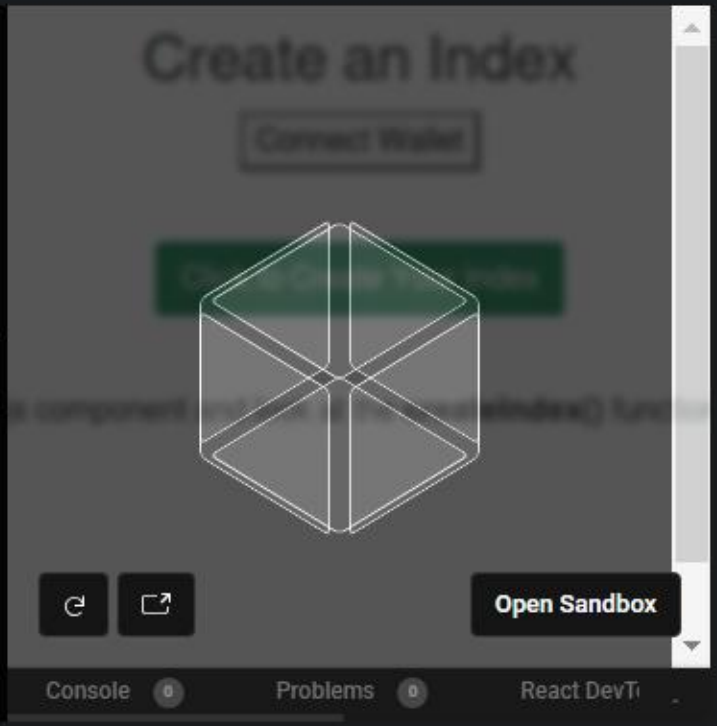
Creating an index will allow you to get started with the IDA. You can use the below example to

generate an index ID which can be used in the next 2 steps. Click the button below to create your index, then open up the console to see your randomly generated index ID.

> NOTE: these examples require test ether/MATIC and test super tokens. You can get some from our faucet here.

CreateIndex.js

```
1   import React, { useState, useEffe
2   import { Framework } from "@super
3   import { Button, Form, Spinner, C
4   import "./createFlow.css";
5   import { ethers } from "ethers";
6
7   let account;
8
9   //where the Superfluid logic take
10  async function createIndex() {
11    const id = Math.floor(Math.rand
12    const provider = new ethers.pro
13    await provider.send("eth_reques
14
15    const signer = provider.getSign
```



Update Your Index By Issuing Units

Once you've created an index, you can issue units to new `subscribers`. You can think of '`units`' as shares within a pool. In the next step, we'll distribute funds to the entire index, which will send a portion of the total amount distributed to each `subscriber`, depending on how many shares each `subscriber` has.

Enter your index ID, a subscriber address, and a number of shares (units) to give your subscriber, then click the button to update your index.

UpdateSubscription.js

```
1  import React, { useState, useEffe
2  import { Framework } from "@super
3  import {
4    Button,
5    Form,
6    FormGroup,
7    FormControl,
8    Spinner,
9    Card
10 } from "react-bootstrap";
11 import "./createFlow.css";
12 import { ethers } from "ethers";
13
14 let account;
15
```

Open Sandbox

Console 0 Problems 0 React DevT

Distribute Funds to Your Index

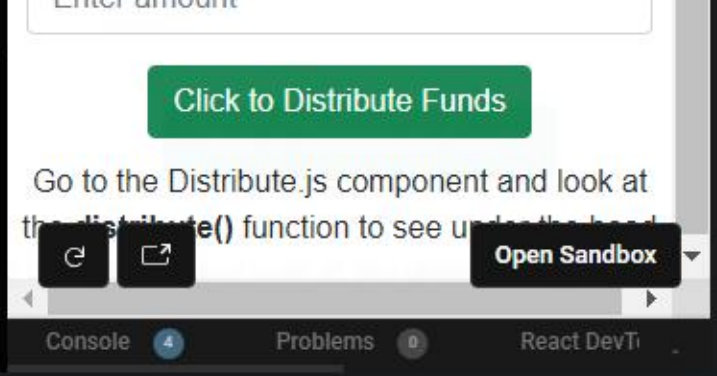
Finally, we can send funds to our index. Enter an amount below in wei, and click the Distribute button to send funds. If you own the address of one of your subscribers, you can visit the 'Distributions' tab in the Superfluid dashboard to approve the subscription & claim that subscriber's portion of the distribution.

The Superfluid protocol is essentially dividing the total amount of funds sent via the distribution according to the amount of units each subscriber holds. You can think of it as similar to shareholders of a company receiving a dividend payment in proportion to the number of shares owned.

Distribute.js

```
import React, { useState, useEffe
import { Framework } from "@super
import {
  Button,
  Form,
  FormGroup,
  FormControl,
  Spinner,
  Card
} from "react-bootstrap";
import "./createFlow.css";
```

```
8     Spinner,
9     Card
10  } from "react-bootstrap";
11  import "./createFlow.css";
12  import { ethers } from "ethers";
13
14  let account;
15
```



Thats it! One thing to keep in mind - for a recipient's balance to reflect the distribution event, they should first call `approveSubscription` one time. If they fail to do this, don't worry, they can still receive their tokens after calling the `claim` function at any time.

If you used an address you have access to in the above tutorial, you can see this process in action by visiting the 'Distributions' tab in the Superfluid dashboard after distributing funds. While there, you'll see the option to either 'approve subscription' or 'claim' previously distributed funds

← Previous
IDA - SDK Core

Next - Developers
Super Apps →

Last modified 6mo ago

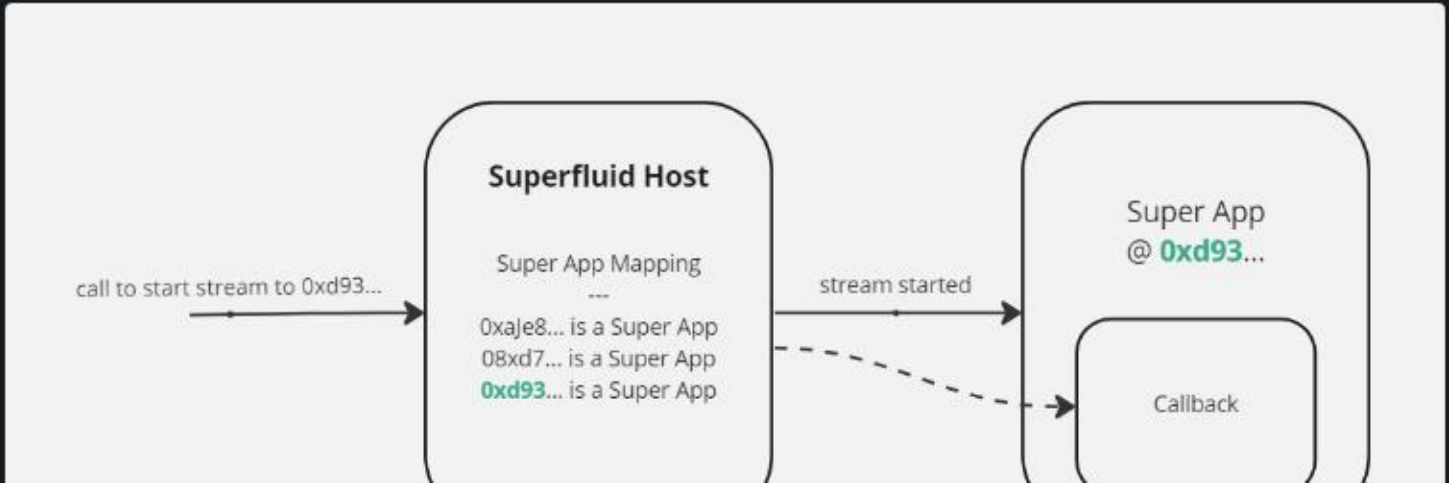
WAS THIS PAGE HELPFUL?

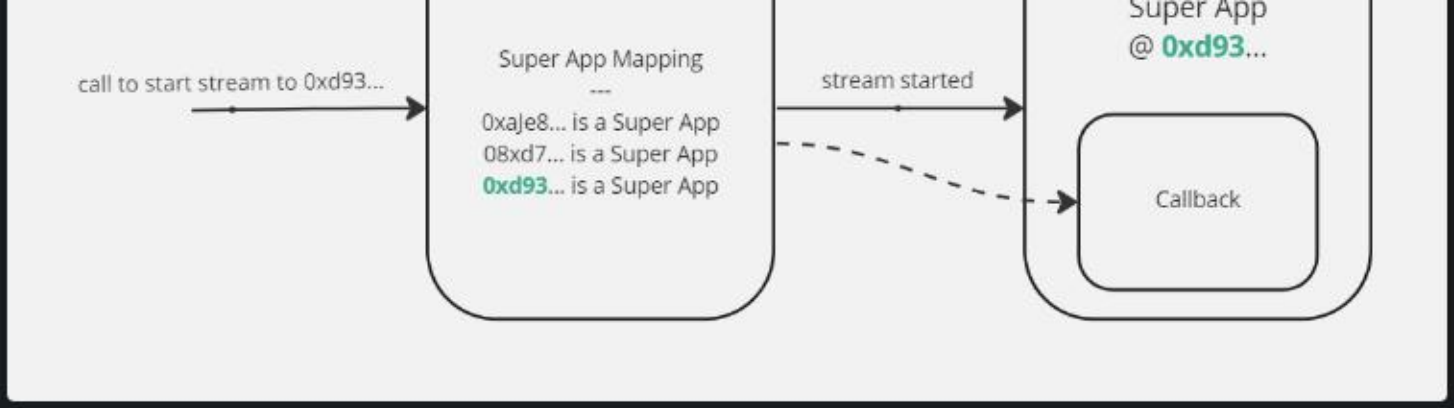
Super Apps

Building Programmable Cashflows

What are Super Apps?

Super Apps are smart contracts that are registered with the Superfluid Protocol allowing them to **react to Super Agreements**. Through callbacks, a Super App can "react" to the creating, updating, and deleting of Super Agreements that it's been engaged with by means of customized logic set in its smart contract code.





This section will show you how to develop code that interacts with the CFA. For something more conceptual, check out this simple conceptual breakdown 👇

Super Apps

Previous
IDA - Frontend Examples

Next
Super Apps in Depth

Last modified 2mo ago

WAS THIS PAGE HELPFUL?

Super Apps in Depth

Using Super Apps To Create Programmable Cashflows

What is a Super App?

Super Apps are smart contracts that are registered with the Superfluid Protocol allowing them to **react to Super Agreements** (like money streams). If you're familiar with ERC777 hooks, they're like those but for Super Agreements.

The Tradeable Cashflow NFT

For example, upon the tradeable cashflow NFT contract receiving a stream, special callbacks

functions within the Super App are triggered which automatically open up a new stream from the NFT contract to the owner of the NFT. The contract 'listens' for a call to a Superfluid super agreement contract (the Constant Flow Agreement contract), and runs a single callback function in response to the following 3 actions:

1. A flow is opened with the Super App as the `receiver`
2. A flow is updated which has the Super App as the `receiver`
3. A flow is closed by the Super App's counter party (i.e. if either the *sender* of a flow into the Super app or the *recipient* of a flow from the Super App deleted the flow, a callback will be run). As of today, this is only relevant for the case of canceled flows.

When a stream is created into a Super App (this will make the Super App the `receiver`), the `beforeAgreementCreated` and the `afterAgreementCreated` callback may be run. These callbacks can execute any arbitrary logic, as long as this logic fits within the rules of standard smart contract development and the rules of Super Apps (which are explained further later on in this section).

In the case of the tradeable cashflow contract, the logic we include inside of the `afterAgreementCreated` callback will open up a money stream from the app to the NFT's owner in an amount that is equal to the `flowRate` into the app.

Super Apps like the tradeable cashflow example keep their callback logic simple, while others get more advanced and leverage items like `userData` for additional functionality.

Some of the most interesting projects in our ecosystem, such as Ricochet Exchange, make heavy

What does it mean for an app to be "jailed" exactly? We apply the term "jailed" to refer to a Super App that failed to comply with the set of rules encoded into the Superfluid framework. This does **not** mean that someone on the Superfluid core team is exerting arbitrary control over your Super Apps.

These rules have been written into the protocol at the software level, and are simply designed to place basic security guardrails on Super Apps. Complex systems in our industry have constraints, and you can think of Super App rules as an extension of those constraints that apply to this subset of the protocol.

> 💡 These rules are a set of restrictions placed on Super Apps that are built into the protocol to protect users. However, these rules are not comprehensive and cannot guarantee that a SuperApp will be 100% safe. Each user should review the Super Apps they are interacting with, and developers should take care to write secure, well-tested Super App code.

Here's an overview of each rule:

1) Super Apps cannot revert in the termination callback (`afterAgreementTerminated()`)

- Use the **trycatch** pattern if performing an action which interacts with other contracts inside of the callback. Doing things like transferring tokens without using the **trycatch** pattern is dangerous and should be avoided.
- Double check internal logic to find **any revert possibility in this callback.**

2) Super Apps can't became insolvent.

- Check if any interaction can lead to **insolvency** situation.
- What is an **insolvency** situation? This occurs when a Super App tries to continue sending funds that it no longer has. Its super token balance must stay > 0 at minimum. You can learn more about liquidation & solvency in our section on this topic.

3) Gas limit operations within the termination callback (`afterAgreementTerminated()`)

- There is a limit of gas limit send in a callback function (*3000000 gas units)*
- If the Super App reverts on terminations calls because of an *out-of-gas* error, it will be jailed.
- For legitimate cases where the app reverts for *out-of-gas* (below the gas limit), the Super App is subject to user decision to send a new transaction with more gas. If the app still reverts, it will be jailed.
- To protect against these cases, **don't create Super Apps that require too much gas within the termination callback**.

4) Incorrect ctx (short for **context)** data within the termination callback

- **Any attempt to tamper with the value of `ctx`** or failing to give the right **`ctx`** will result in a Jailed App.
- Any time a protocol function returns a `ctx`, that `ctx` should be passed to the next called function. It will repeat this process even in the return of the callback itself.
- For more information on `ctx` and how it works you can check out our tutorial on userData.

be jailed.

- To protect against these cases, **don't create Super Apps that require too much gas within the termination callback**.

4) Incorrect ctx (short for **context)** data within the termination callback

- **Any attempt to tamper with the value of `ctx`** or failing to give the right `ctx` will result in a Jailed App.
- Any time a protocol function returns a `ctx`, that `ctx` should be passed to the next called function. It will repeat this process even in the return of the callback itself.
- For more information on `ctx` and how it works you can check out our tutorial on userData.

For good Super App examples, you can head to our examples repo, as well as the Super App tutorial and User Data tutorial. If you have a Super App you'd like to build, please reach out to us in the #developers channel of our discord and someone on our team will be there to help you get started.

← Developers - Previous
Super Apps

Next
Super App Callbacks →

use of Super App callbacks.

Super App Configuration

For a Super App to be able to use callbacks, it must first be 'registered' with the protocol. This is done using a pattern similar to the ERC1820 registry system.

To register a Super App, you need to add the following code to your Super App's constructor:

```
// by default, all 6 callbacks defined in the ISuperApp interface
// are forwarded to a SuperApp.
// If you inherit from SuperAppBase, there's a default implementation
// for each callback which will revert.
// Developers will want to avoid reverting in Super App callbacks,
// In particular, you want to avoid reverting within the termination callback
// (see rules below regarding the termination callback for more info)
// you need to make sure only those actually implemented (overridden)
// are ever invoked. That's achieved by setting the _NOOP flag for those
// callbacks which you don't need and didn't implement.
uint256 configWord =
            SuperAppDefinitions.APP_LEVEL_FINAL |
            SuperAppDefinitions.BEFORE_AGREEMENT_CREATED_NOOP |
            SuperAppDefinitions.BEFORE_AGREEMENT_UPDATED_NOOP |
            SuperAppDefinitions.BEFORE_AGREEMENT_TERMINATED_NOOP;

// can be an empty string in dev or testnet deployments
string memory registrationKey = "";
```

```
        _host.registerAppWithKey(configWord, registrationKey);
```

Running `_host.registerAppWithKey` and passing in the `configWord` will enable your Super App to be registered within the Superfluid host contract's Super App manifest. This will allow it to be managed under basic Superfluid governance parameters and ensure that callbacks are run as intended.

The `APP_LEVEL_FINAL` must be set as seen above for now. This parameter refers to which (and how many) callbacks will be run within a hypothetical chain of Super Apps.

For example, if a user calls a Super app, then that app calls another Super App, do the Super App callbacks run in both contracts, the second contract in the chain of events, or the first contract in the chain of events? The answer to this question, as of Q4 2021, is that the callbacks are run within the **first app only.**

The `_NOOP` (pronounced 'no-op') designations are also important as they allow you to specify which callbacks you'd like to use in your Super App. Callbacks can run **before** and/or **after** an agreement function runs.

If you inherit from the SuperAppBase contract, there's a default implementation for each callback which will revert. It's critical to ensure you have a well-tested implementation for each callback, and that you use the `_NOOP` pattern to prevent the callbacks you *don't* wish to use from running. In the case of an app that only uses the `afterAgreement` callbacks, you'll want to add the `_NOOP` flag to each callback you won't be using - in this case the `beforeAgreement` callbacks. This can be done like so within your `configWord` variable.

```
SuperAppDefinitions.BEFORE_AGREEMENT_CREATED_NOOP |
SuperAppDefinitions.BEFORE_AGREEMENT_UPDATED_NOOP |
SuperAppDefinitions.BEFORE_AGREEMENT_TERMINATED_NOOP;
```

Finally, you can register your app using `_host.registerAppWithKey(configWord)` or with `_host.registerAppByFactory(app, configWord)`.

You can do this freely on testnets, but on mainnets pre-approval of Super Apps is needed for now. Navigate to this guide in order to find out how to get that done.

Super App CFA Buffers

When a user first creates a Superfluid stream, the protocol will take an up front buffer to ensure the protocol's security. These deposits are sized as follows:

- On testnets, when the flow is being sent to a recipient that is *not* a Super App, this amount is 1hr x the `flowRate`
- On testnets, when the flow is being sent to a recipient that *is* a Super App, the deposit amount can be up to 2hrs x the `flowRate`
- On mainnet, when the flow is being sent to a recipient that is *not* a Super App, the deposit amount is 4hrs x the `flowRate`
- On mainnet, when the flow is being sent to a recipient that *is* a Super App, the deposit can be up to 8hrs x the `flowRate` (note that this is approximate)

Why can the deposit by up to ~2x when sending a stream to a Super App?

Because Super Apps are designed to enable programmable cashflows, the protocol needs to ensure that Super Apps don't execute logic that could impact the protocol's security. The caller of the stream being sent to the super app is essentially *covering the deposit* of the Super App.

This higher deposit provides an extra incentive for the protocol to ensure that users sending streams into super apps maintain a balance > 0.

Super App Callbacks

Super App callbacks are run when a Super App is on the receiving end of a callAgreement call (i.e. an operation which executes logic in one of the existing agreement contracts, such as the IDA or CFA)

When Will Super App Callbacks Run?

In the case of the constant flow agreement, Super App callbacks will run when

- a stream is created where the app is the `receiver` of the stream.
- a stream is updated where the app is the `receiver` of the stream.
- a stream is deleted by a user or contract that is external to the Super App.

Note that in the third case, a stream may be deleted by *either the sender or the receiver* of that stream. This means that the `beforeAgreementTerminated` or `afterAgreementTerminated`

stream. This means that the `beforeAgreementTerminated` or `afterAgreementTerminated` callback will run if a stream that was being sent into the Super App was deleted by the sender, or if a stream being sent from the Super App to another address was deleted by the receiver of that stream.

Who is Calling the Super App Callback?

Callbacks are called entirely on chain by the Superfluid protocol in response to events. Each time an action is taken inside of the Constant Flow Agreement contract, the protocol will check the Super App registry to determine whether or not the stream will be sent to a Super App (or if the stream being deleted involves a Super App in the event of deletion). If it doesn't involve a Super App, there are no callbacks to run, so the stream is created, updated, or deleted without any other operation.

However, if the protocol finds that the stream indeed does involve a Super App after a check to the Superfluid Super App manifest, it will call the necessary callback(s).

The Anatomy of Super App Callbacks:

```
function beforeAgreementCreated(
        ISuperToken /*superToken*/,
        address /*agreementClass*/,
        bytes32 /*agreementId*/,
        bytes calldata /*agreementData*/,
        bytes calldata /*ctx*/
    )
        external
        view
```

```
    view
    virtual
    override
    returns (bytes memory /*cbdata*/)
{
    revert("Unsupported callback - Before Agreement Created");
}
```

```
function afterAgreementCreated(
    ISuperToken /*superToken*/,
    address /*agreementClass*/,
    bytes32 /*agreementId*/,
    bytes calldata /*agreementData*/,
    bytes calldata /*cbdata*/,
    bytes calldata /*ctx*/
)
    external
    virtual
    override
    returns (bytes memory /*newCtx*/)
{
    revert("Unsupported callback - After Agreement Created");
}
```

A `beforeAgreement` callback will be run before the call to the agreement contract will be run. For example, if there is logic inside of the `beforeAgreementCreated` callback within of a Super App,

and a user opens a stream into that Super App contract, the logic inside of `beforeAgreementCreated` will run before the stream is created.

Similarly, an `afterAgreement` callback will be run after the call to the agreement contract is run. For example, if there is logic inside of the `afterAgreementCreated` callback within a Super App, and a user opens a stream into that Super App contract, the logic inside of `afterAgreementCreated` will run after the stream is created.

One additional thing to note about the `beforeAgreement` callbacks is that they are `view` functions. So, if you want to, for example, save a variable to state in response to something that happens in the `beforeAgreement` callback, you should do the following:

1. Return the data that you want to save inside the `beforeAgreement` callback (this returned value will be passed to the `afterAgreement` callback as `cbdata`, which we explain below)
2. Save the variable to state inside of the `afterAgreement` callback
3. Make sure that you have an implementation for both the `beforeAgreement` and `afterAgreement` callbacks

Breaking Down Each Variable

`ISuperToken` - the protocol will pass the Super Token that's being used in the call to the constant flow agreement contract here.

`address` - this will be the address of the Constant Flow Agreement contract on the network you're interacting with. You can find more details around these networks inside of the Superfluid network directory.

address - this will be the address of the Constant Flow Agreement contract on the network you're interacting with. You can find more details around these networks inside of the Superfluid network directory.

agreementId - a bytes32 value that is a hash of the sender and receiver's address of the flow that was created, updated, or deleted.

agreementData - the address of the sender and receiver of the flow that was created, updated, or deleted - encoded using solidity's `abi.encode()` built in function

cbdata - this contains data that was returned by the `beforeAgreement` callback *if* it was run prior to the calling of `afterAgreement` callback. **Note**: this is only passed to the `afterAgreement` callback

ctx - this contains data about the call to the Constant Flow Agreement contract itself. 'Ctx' is short for 'context' and is explained in depth inside of our tutorial on userData (which you can access inside of the `ctx` value).

Super App Rules (Jail System)

Super Apps are a powerful concept within the Superfluid ecosystem. They allow for new levels of creativity - specifically related to *programmable cash flows.*

However, there are specific rules that have been encoded into the protocol which SuperApps must abide by. Super App rules should be obeyed at all cost by developers, or they risk their contract being jailed by the protocol.

What does it mean for an app to be 'jailed' exactly? We apply the term `jailed` to refer to a Super

Super App Callbacks

Making use of hooks that run before or after an agreement is called

Super App Callbacks

Super App callbacks are run when a Super App is on the receiving end of a transaction that creates, updates, or deletes an agreement in relation to that app.

When Will Super App Callbacks Run?

The Constant Flow Agreement and Instant Distribution Agreement activate different callbacks in different scenarios. In the following table, the Constant Flow Agreement is referred to as CFAv1 and the Instant Distribution Agreement is referred to as IDAv1.

Agreement	Callback	Condition
CFAv1	beforeAgreementCreated, afterAgreementCreated	A stream to a Super App is created.
CFAv1	beforeAgreementUpdated, afterAgreementUpdated	A stream to a Super App is updated.

CFAv1	beforeAgreementTerminated, afterAgreementTerminated	A stream to a Super App is deleted.
IDAv1	beforeAgreementCreated, afterAgreementCreated	A subscription (with zero units) to an index published by a Super App is approved.
IDAv1	beforeAgreementUpdated, afterAgreementUpdated	A subscription (with units) to an index published by a Super App is approved.
IDAv1	beforeAgreementTerminated, afterAgreementTerminated	A subscription to an index published by a Super App is revoked.
IDAv1	beforeAgreementUpdated, afterAgreementUpdated	A subscription to an index published by a Super App is claimed
IDAv1	beforeAgreementCreated, afterAgreementCreated	Units of an index are issued to a Super App if the units were previously zero.
IDAv1	beforeAgreementUpdated, afterAgreementUpdated	Units of an index are issued to a Super App if the units were previously non-

	afterAgreementUpdated	units were previously non zero.
IDAv1	beforeAgreementTerminated, afterAgreementTerminated	Units of an index issued to a Super App are deleted.

Who is Calling the Super App Callback?

Callbacks are called entirely on chain by the Superfluid protocol in response to events. Each time an action is taken inside of the Constant Flow Agreement contract, the protocol will check the Super App registry to determine whether or not the stream will be sent to a Super App (or if the stream being deleted involves a Super App in the event of deletion). If it doesn't involve a Super App, there are no callbacks to run, so the stream is created, updated, or deleted without any other operation. However, if the protocol finds that the stream indeed does involve a Super App after a check to the Superfluid Super App manifest, it will call the necessary callback(s).

The Anatomy of Super App Callbacks:

```
function beforeAgreementCreated(
        ISuperToken /*superToken*/,
        address /*agreementClass*/,
        bytes32 /*agreementId*/,
        bytes calldata /*agreementData*/,
        bytes calldata /*ctx*/
    )
        external
```

```
        view
        virtual
        override
        returns (bytes memory /*cbdata*/)
    {
        revert("Unsupported callback - Before Agreement Created");
    }
```

```
    function afterAgreementCreated(
        ISuperToken /*superToken*/,
        address /*agreementClass*/,
        bytes32 /*agreementId*/,
        bytes calldata /*agreementData*/,
        bytes calldata /*cbdata*/,
        bytes calldata /*ctx*/
    )
        external
        virtual
        override
        returns (bytes memory /*newCtx*/)
    {
        revert("Unsupported callback - After Agreement Created");
    }
```

A `beforeAgreement` callback will be run before the call to the agreement contract will be run. For

example, if there is logic inside of the `beforeAgreementCreated` callback within of a Super App, and a user opens a stream into that Super App contract, the logic inside of `beforeAgreementCreated` will run before the stream is created.

Similarly, an `afterAgreement` callback will be run after the call to the agreement contract is run. For example, if there is logic inside of the `afterAgreementCreated` callback within a Super App, and a user opens a stream into that Super App contract, the logic inside of `afterAgreementCreated` will run after the stream is created.

One additional thing to note about the `beforeAgreement` callbacks is that they are `view` functions. So, if you want to, for example, save a variable to state in response to something that happens in the `beforeAgreement` callback, you should do the following:

1. Return the data that you want to save inside the `beforeAgreement` callback (this returned value will be passed to the `afterAgreement` callback as `cbdata`, which we explain below)
2. Save the variable to state inside of the `afterAgreement` callback
3. Make sure that you have an implementation for both the `beforeAgreement` and `afterAgreement` callbacks

Breaking Down Each Variable

`ISuperToken` - the protocol will pass the Super Token that's being used in the call to the constant flow agreement contract here.

`address` - this will be the address of the Constant Flow Agreement contract on the network you're interacting with. You can find more details around these networks inside of the Superfluid network

directory.

`agreementId` - a bytes32 value that is a hash of the sender and receiver's address of the flow that was created, updated, or deleted.

`agreementData` - the address of the sender and receiver of the flow that was created, updated, or deleted - encoded using solidity's `abi.encode()` built in function

`cbdata` - this contains data that was returned by the `beforeAgreement` callback *if* it was run prior to the calling of `afterAgreement` callback. **Note**: this is only passed to the `afterAgreement` callback

`ctx` - this contains data about the call to the constant flow agreement contract itself. 'Ctx' is short for 'context' and is explained in depth inside of our tutorial on userData (which you can access inside of the `ctx` value).

← Previous
Super Apps in Depth

Next
Calling Agreements In Super Apps →

directory.

`agreementId` - a bytes32 value that is a hash of the sender and receiver's address of the flow that was created, updated, or deleted.

`agreementData` - the address of the sender and receiver of the flow that was created, updated, or deleted - encoded using solidity's `abi.encode()` built in function

`cbdata` - this contains data that was returned by the `beforeAgreement` callback *if* it was run prior to the calling of `afterAgreement` callback. **Note**: this is only passed to the `afterAgreement` callback

`ctx` - this contains data about the call to the constant flow agreement contract itself. 'Ctx' is short for 'context' and is explained in depth inside of our tutorial on userData (which you can access inside of the `ctx` value).

← Previous
Super Apps in Depth

Next
Calling Agreements In Super Apps →

Calling Agreements In Super Apps

Understanding CallAgreement vs CallAgreementWithContext

> Note that this is a highly technical section for those looking to understand lower level features of the protocol. If you're looking to create, update, and delete streams or work with the instant distribution agreement in your super app callbacks, you can refer to the SuperTokenV1Library to perform these operations in a single line of code.

CallAgreement vs CallAgreementWithContext

TLDR

If you're making a call to a Superfluid agreement inside of a Super App callback, you should remember that you need to "receive a context, and return a context" by using the `callAgreementWithContext` function. If you're not making this call inside of a Super App callback, you should use `callAgreement`.

In Depth

When calling the host contract to trigger actions related to the constant flow agreement (CFA) or instant distribution agreement (IDA), you may use either `callAgreement` or `callAgreementWithContext`. Both of these functions allow you to pass in an encoded call to the agreement you'd like to interact with, as well as an optional `userData` value. The `callAgreementWithContext` function will perform the same actions as the `callAgreement` function, but it also enables you to pass in a new context value (abbreviated `ctx`) to your function call.

Keep in mind that `callAgreementWithContext` is designed for use within Super Apps - if this function is run outside of a Super App, then `callAgreementWithContext` will fail due to this statement:

```
require(
    context.appAddress == msg.sender,
    "SF: callAgreementWithContext from wrong address"
);
```

`callAgreementWithContext` is primarily meant for use within Super App callbacks. Each Super App callback will be passed a context value (`ctx`) from the host contract (as the Superfluid host contract is the *caller* of each callback). This `ctx` value is what needs to be passed to any call you want to make to the Superfluid host contract *inside* of your callback (this goes for all operations which create, update, and delete flows in these callbacks). As a reminder, the logic within the 'host' contract can be found in `Superfluid.sol`.

The process looks like this:

```
//a function which we'll use to create a flow within a callback
function _createFlowInCallback(
        bytes calldata ctx,
        ISuperfluid _host,
        IConstantFlowAgreementV1 _cfa,
        ISuperfluidToken _acceptedToken,
        address _receiver,
        int96 _flowRate
    )
        private
        returns (bytes memory newCtx)
    {
        newCtx = ctx;

        (newCtx,) = _host.callAgreementwithContext(
            _cfa,
            abi.encodeWithSelector(
            _cfa.deleteFlow.selector,
            _acceptedToken,
            address(this),
            _receiver,
            new bytes(0) // placeholder
          ),
          "0x", //placeholder userdata value
          newCtx //passing in the context from the super app callback
        );
```

```
        );
}

function afterAgreementCreated(
    ISuperToken _superToken,
    address _agreementClass,
    bytes32, // _agreementId,
    bytes calldata /*_agreementData*/,
    bytes calldata ,// _cbdata,
    bytes calldata _ctx
)
    external override
    onlyExpected(_superToken, _agreementClass)
    onlyHost
    returns (bytes memory newCtx)
{
//passing in the ctx which is sent to the callback here
   return _createFlowInCallback(_ctx, _host, _cfa, _acceptedToken, _receiver, _flowrate
}
```

You can also do this in a much easier way by using our new CFA Library, which abstracts away the need to use `host.callAgreement` or `host.callAgreementWithContext` directly.

```
using CFALibraryV1 for CFALibraryV1.InitData;

    //initialize cfaV1 variable
```

```
    CFALibraryV1.InitData public cfaV1;

    constructor(
        ISuperfluid host
    ) {

    //initialize InitData struct, and set equal to cfaV1
    cfaV1 = CFALibraryV1.InitData(
       host,
       IConstantFlowAgreementV1(
          address(host.getAgreementClass(
           keccak256("org.superfluid-finance.agreements.ConstantFlowAgreement.v1")
           ))
        )
     );
   }

function afterAgreementCreated(
    ISuperToken _superToken,
    address _agreementClass,
    bytes32, // _agreementId,
    bytes calldata /*_agreementData*/,
    bytes calldata ,// _cbdata,
    bytes calldata _ctx
)
    external override
    onlyExpected(_superToken, _agreementClass)
```

```
    onlyHost
    returns (bytes memory newCtx)
{
//passing in the ctx which is sent to the callback here
//createFlowWithCtx makes use of callAgreementWithContext
   return cfaV1.createFlowWithCtx(_ctx, receiver, token, flowRate);
}
```

You may have another scenario in which you want to make additional calls to the host contract after you first run `callAgreementWithContext`. If you do this, you can save the value returned by the first `callAgreementWithContext` function to a new variable, then pass this value to your next call to `callAgreementWithContext`. The takeaway here is that you need to pass the most recent iteration of `ctx` when creating, updating, or deleting flows inside Super App callbacks. You can see this done here with the CFA Library:

```
using CFALibraryV1 for CFALibraryV1.InitData;

    //initialize cfaV1 variable
    CFALibraryV1.InitData public cfaV1;

    constructor(
        ISuperfluid host
    ) {

    //initialize InitData struct, and set equal to cfaV1
    cfaV1 = CFALibraryV1.InitData(
```

```
        host,
        IConstantFlowAgreementV1(
           address(host.getAgreementClass(
            keccak256("org.superfluid-finance.agreements.ConstantFlowAgreement.v1")
             ))
         )
      );
    }

function afterAgreementCreated(
    ISuperToken _superToken,
    address _agreementClass,
    bytes32, // _agreementId,
    bytes calldata /*_agreementData*/,
    bytes calldata ,// _cbdata,
    bytes calldata _ctx
)
    external override
    onlyExpected(_superToken, _agreementClass)
    onlyHost
    returns (bytes memory newCtx)
{
   newCtx = cfaV1.createFlowWithCtx(_ctx, receiver, token, flowRate); //passing in the
   newCtx = cfaV1.createFlowWithCtx(newCtx, receiver, token, flowRate); //passing in the

}
```

One final item to note is to not manually change the value of `ctx` when it's used within a Super App. `Ctx` is formatted in a very specific way within a struct that is compiled to bytecode by the protocol, and it's not meant to be manipulated directly. If you wish to pass in userData to your function call, this can be done by simply adding the `userData` value in as a parameter:

```
//using the CFA Library:
function afterAgreementCreated(
    ISuperToken _superToken,
    address _agreementClass,
    bytes32, // _agreementId,
    bytes calldata /*_agreementData*/,
    bytes calldata ,// _cbdata,
    bytes calldata _ctx
)
    external override
    onlyExpected(_superToken, _agreementClass)
    onlyHost
    returns (bytes memory newCtx)
{
   //passing in the ctx which is sent to the callback here
   //createFlowWithCtx makes use of callAgreementWithContext
   return cfaV1.createFlowWithCtx(_ctx, receiver, token, flowRate, userData);
}

//using a low level call
```

```
function afterAgreementCreated(
    ISuperToken _superToken,
    address _agreementClass,
    bytes32, // _agreementId,
    bytes calldata /*_agreementData*/,
    bytes calldata ,// _cbdata,
    bytes calldata _ctx
)
    external override
    onlyExpected(_superToken, _agreementClass)
    onlyHost
    returns (bytes memory newCtx)
  {
        newCtx = _ctx;
         (newCtx,) = _host.callAgreementwithContext(
              _cfa,
              abi.encodeWithSelector(
              _cfa.deleteFlow.selector,
              _acceptedToken,
              address(this),
              _receiver,
        new bytes(0) // placeholder
      ),
      userData, //userData goes here
      newCtx //passing in the context from the super app callback
   );
 }
```

```
            _receiver,
        new bytes(0) // placeholder
      ),
      userData, //userData goes here
      newCtx //passing in the context from the super app callback
    );
  }
```

If you read through the Superfluid codebase, you'll see that nearly every state changing operation will return a context value. This `ctx` value helps to provide additional internal accounting for the protocol to enhance security, and it allows you to decode it and make use of values like userData inside Super Apps. When making calls within your Super Apps, keep in mind that you need to pass in updated context values if you want to make use of the callbacks properly. Remember: if you need to run callAgreement within a Super App callback, you'll need to use `callAgreementWithContext` and pass in `ctx`.

← Previous
Super App Callbacks

Next
User Data →

User Data

Including additional metadata in your Super Apps

Another powerful component of the Superfluid protocol is the ability to pass in additional user data along with your calls to super agreements. Think of it like metadata that can accompany your streams or IDAs 😎

Before we look at user data, let's take a quick dive into a new element: **Context**.

Context is used for several key items within the Superfluid protocol such as gas optimization, governance, security, and SuperApp callbacks. One parameter that's also available for use within the context field is userData.

This is from the host interface (ISuperfluid.sol) file inside of our interfaces folder in the Superfluid repo. On line 20, we see `userData`.

```
1 struct Context {
2     //
3     // Call context
4     //
5     // callback level
6     uint8 appLevel;
```

```
    // type of call
    uint8 callType;
    // the system timestsamp
    uint256 timestamp;
    // The intended message sender for the call
    address msgSender;

    //
    // Callback context
    //
    // For callbacks it is used to know which agreement function selector is called
    bytes4 agreementSelector;
    // User provided data for app callbacks
    bytes userData;

    //
    // App context
    //
    // app allowance granted
    uint256 appAllowanceGranted;
    // app allowance wanted by the app callback
    uint256 appAllowanceWanted;
    // app allowance used, allowing negative values over a callback session
    int256 appAllowanceUsed;
    // app address
    address appAddress;
    // app allowance in super token
    ISuperfluidToken appAllowanceToken;
```

```
    ISuperfluidToken appAllowanceToken;
}
```

Whenever you see `ctx` being moved around within the protocol, this struct is what's under the hood (it's just compiled down to bytes each time it's passed between functions).

As you can see, `userData` is one of the elements that makes up `Context`. For the sake of this tutorial, we're going to focus exclusively on **`userData`** for the time being.

Quick Review: How Are Super Agreements Called Again?

To call a function in a Super Agreement, you first need to use `abi.encode` to compile a function call to the super agreement you're looking to trigger. Then, you need to pass the agreement type, the bytecode of the previously compiled function call, and `userData` to `callAgreement` (we'll get to userData next). The whole process looks like this:

```
// Addresses for host and cfa on Polygon
ISuperfluid host = "0x3E14dC1b13c488a8d5D310918780c983bD5982E7";
IConstantFlowAgreementV1 cfa = "0x6EeE6060f715257b970700bc2656De21dEdF074C";
// DAIx
ISuperToken acceptedToken = "0x8f3cf7ad23cd3cadbd9735aff958023239c6a063";solidss
// empty user data
bytes userData = "0x";

// $1000 DAI per month
```

```
int96 flowRate = "385802469135802";

// receiver is arbitrary
address receiver = "0x...";

host.callAgreement(
     cfa,
     abi.encodeWithSelector(
         cfa.createFlow.selector,
         acceptedToken,
         receiver
         flowRate,
         new bytes(0) // placeholder
     ),
     userData,
);
```

> **Note**: `userData` is always passed into `callAgreement` as type `bytes`.

```
// call agreement interface
function callAgreement(
   ISuperAgreement agreementClass,
   bytes calldata callData,
   bytes calldata userData
)
```

```
    external
    //cleanCtx
    //isAgreement(agreementClass)
    returns(bytes memory returnedData);
```

Behind the scenes, your `userData` variable is appended onto `Context`, which is then available to you as a developer in the SuperApp callbacks.

When you execute an operation in the CFA contract for example (and create, update, or delete a flow), you'll have access to the Context that's available after the initial call to the protocol was made. For example, if I pass in `userData` when a create a flow into a Super App, I can decode the `context` & this user data inside any of the super app callbacks, and re-use or manipulate this data as I please. For example, if I send a transaction where `receiver` is a SuperApp, and pass along an encoded string 'hello sir' as `userData`:

```
string unformattedUserData = 'hello sir';
bytes userData = abi.encode(unformattedUserData);

host.callAgreement(
    cfa,
    abi.encodeWithSelector(
        cfa.createFlow.selector,
        acceptedToken,
        //receiver is a super app...
        receiver
        flowRate,
```

```
        new bytes(0) // placeholder
    ),
    userData,
);
```

I can decode the context that's passed into the callback, which will give me the Context struct displayed above. Then, since userData is one of the fields on the struct, we can abi.decode userData get back my value of 'hello sir' on the other side:

```
// inside of the afterAgreementCreated Super App Callback

function afterAgreementCreated(
    ISuperToken _superToken,
    address _agreementClass,
    bytes32, // _agreementId,
    bytes calldata /*_agreementData*/,
    bytes calldata ,// _cbdata,
    bytes calldata _ctx
)
    external override
    onlyExpected(_superToken, _agreementClass)
    onlyHost
    returns (bytes memory newCtx)
{

    // decode Contex - this will return the entire Context struct
```

```
    ISuperfluid.Context memory decompiledContext = _host.decodeCtx(_ctx);

    // userData is a one of the fields on the Context struct
    //set decodedUserData variable to decoded value
    // this will return 'hello sir'
    decodedUserData = abi.decode(decompiledContext.userData, (string));

    // do some stuff with your decodedUserData
    return _doSomeStuff(decodedUserData);
}
```

In Conclusion

UserData can be *any* arbitrary piece of data. Think of it as metadata that's associated with anything you do in a Super Agreement.

This metadata could be used for a wide variety of use cases:

- You could pass in data to accompany a salary or payment stream - perhaps employee info or product info
- You can send a message along with your distribution in an instant distribution agreement
- You could even pass in the byte code for another entire smart contract.

We invite you to be creative with this!

Next up: a tutorial on how to leverage UserData within your applications

In Conclusion

UserData can be *any* arbitrary piece of data. Think of it as metadata that's associated with anything you do in a Super Agreement.

This metadata could be used for a wide variety of use cases:

- You could pass in data to accompany a salary or payment stream - perhaps employee info or product info
- You can send a message along with your distribution in an instant distribution agreement
- You could even pass in the byte code for another entire smart contract.

We invite you to be creative with this!

Next up: a tutorial on how to leverage UserData within your applications.

← Previous
Calling Agreements In Super Apps

Next
NFT Billboard Example →

NFT Billboard Example

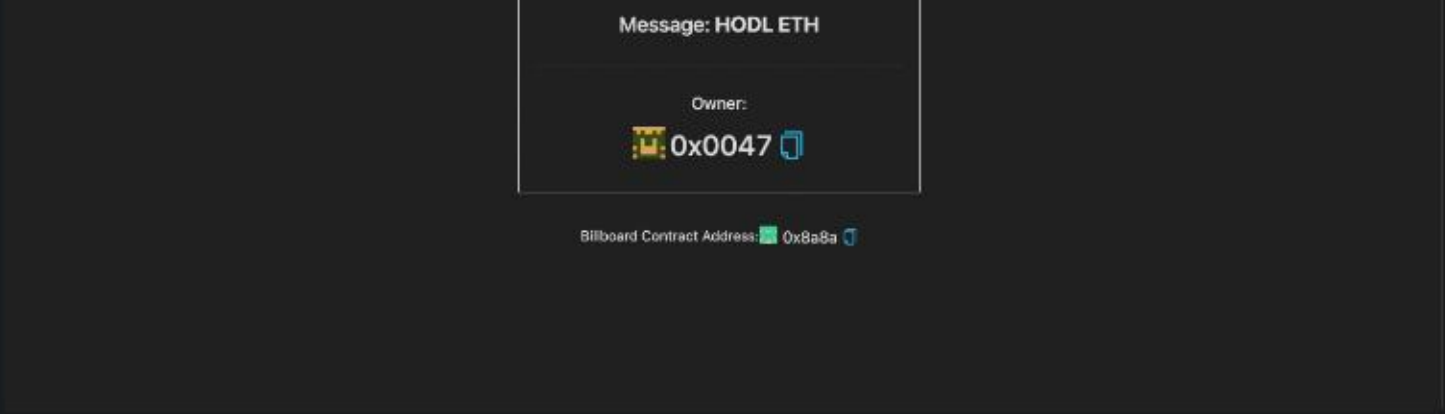
Create Your Own NFT Billboard with User Data

Build an NFT Billboard with Superfluid UserData

In this tutorial, we make a small tweak to the contracts used in the TradeableCashflow Super App to test out Superfluid's userData parameters. We set up a Scaffold-eth based repo, remove (comment out) some of the extra front end stuff, and then create a few scripts to allow us to easily create, read, update, and delete flows (we love CRUD).

> NOTE: We recommend reading through our Super Apps tutorial before completing this tutorial.

Our dapp will turn the TradeableCashflow into a tradeable NFT billboard that can be rented with streams. The message displayed on our billboard will be the parameter passed in as userData. If the billboard is traded, all rental cashflows will be redirected toward the new owner.



This tutorial is most certainly NOT investment advice, but we'll assume that someone out there will want others to HODL their favorite assets...a decent use case for a billboard

You can follow along with the video version of this tutorial on Youtube, and fork the repo here as well:

protocol-monorepo/examples/nftbillboard-userdata at chainshot · superfluid-finance/...
GitHub

Contract Deployment

Scaffold-eth makes contract deployment easy: we simply need to use the already installed hardhat plugin, hardhat-deploy, to run our deployments. Each time we want to deploy a new NFT billboard contract, we can simply run `yarn deploy` to do so.

Inside of the already created deploy folder from Scaffold-eth, we'll pass in a few key addresses from the superfluid protocol as variables, and use them to write a deployment script for our Tradeable Cashflow (NFT billboard) contract.

We get the addresses for `host`, `cfa`, and `fDAIx` from the Superfluid network directory. If you want to deploy to a different testnet, you'll need to make adjustments here.

```
//inside of deploy/00_deploy_tradeable_cashflow.js
//mumbai addresses - change if using a different network

const host = '0xEB796bdb90fFA0f28255275e16936D25d3418603';
const cfa = '0x49e565Ed1bdc17F3d220f72DF0857C26FA83F873';
const fDAIx = '0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f';
//your address here...
const owner = '0x...';
```

Next, we'll need to add a deployment script for our TradeableCashflow.sol contract. Remember, TradeableCashflow.sol inherits from RedirectAll.sol, so we only need to deploy the single NFT contract.

We'll use the function getNamedAccounts() to get our deployer address. Because we included our

We'll use the function getNamedAccounts() to get our deployer address. Because we included our account in the 'accounts' property in 'polytest' within hardhat.config, we'll get our address first when calling that function.

We can then deploy our contract fromm that address, and pass in the necessary parameters to the constructor of the TradeableCashflow contract (owner, name, symbol, host, cfa, acceptedToken).

```
//inside of deploy/00_deploy_tradeable_cashflow.js
module.exports = async ({ getNamedAccounts, deployments }) => {
  const { deploy } = deployments;

  const { deployer } = await getNamedAccounts();
  console.log(deployer);

  await deploy("TradeableCashflow", {
    from: deployer,
    args: [owner, 'nifty_billboard', 'NFTBoard', host, cfa, fDAIx],
    log: true,
  })

//...

module.exports.tags = ["YourContract"];
```

Hardhat Scripts For Flow CRUD Functionality

```
//at the top of each script
const hre = require("hardhat");
require("dotenv");
const Web3 = require("web3");

//all addresses hardcoded for mumbai
const hostJSON = require("../artifacts/@superfluid-finance/ethereum-contracts/contracts,
const hostABI = hostJSON.abi;
const hostAddress = "0xEB796bdb90fFA0f28255275e16936D25d3418603";

const cfaJSON = require("../artifacts/@superfluid-finance/ethereum-contracts/contracts/
const cfaABI = cfaJSON.abi;
const cfaAddress = "0x49e565Ed1bdc17F3d220f72DF0857C26FA83F873";

const tradeableCashflowJSON = require("../artifacts/contracts/TradeableCashflow.sol/Trac
const tradeableCashflowABI = tradeableCashflowJSON.abi;

//temporarily hardcode contract address and sender address
//need to manually enter contract address and sender address here
const deployedTradeableCashflow = require("../deployments/polytest/TradeableCashflow.jsc
const tradeableCashflowAddress = deployedTradeableCashflow.address;

//your address here
const _sender = "0x...";
```

Then, in the bulk of our createFlow, updateFlow, and deleteFlow scripts, we'll create contract objects, define a web3 provider, and submit our call using the pattern outlined in Alchemy's docs on submitting transactions with web3.js.

```
//create a flow
async function main() {


  const web3 = new Web3(new Web3.providers.HttpProvider(process.env.MUMBAI_ALCHEMY_URL)

  //create contract instances for each of these
  const host = new web3.eth.Contract(hostABI, hostAddress);
  const cfa = new web3.eth.Contract(cfaABI, cfaAddress);
  const tradeableCashflow = new web3.eth.Contract(tradeableCashflowABI, tradeableCashfl

  const fDAIx = "0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f"
  const userData = web3.eth.abi.encodeParameter('string', 'HODL BTC');

  const nonce = await web3.eth.getTransactionCount(_sender, 'latest'); // nonce starts

  //create flow by calling host directly in this function
  //create flow from sender to tradeable cashflow address
  //pass in userData to the flow as a parameter
  async function startFlow() {
      let cfaTx = (await cfa.methods
     .createFlow(
      fDAIx,
```

```
    // _sender,
    tradeableCashflowAddress,
    "3858024691358",
    "0x"
   )
   .encodeABI())

   let txData = (await host.methods.callAgreement(
    cfaAddress,
    cfaTx,
    userData
  ).encodeABI());

  let tx = {
    'to': hostAddress,
    'gas': 3000000,
    'nonce': nonce,
    'data': txData
  }

  let signedTx = await web3.eth.accounts.signTransaction(tx, process.env.MUMBAI_DEPLO

  await web3.eth.sendSignedTransaction(signedTx.rawTransaction, function(error, hash)
    if (!error) {
      console.log("🎉 The hash of your transaction is: ", hash, "\n Check Alchemy's M
    } else {
      console.log("❗Something went wrong while submitting your transaction:", error)
```

```
      }
     });

    }


  await startFlow();

  }

// We recommend this pattern to be able to use async/await everywhere
// and properly handle errors.
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

Updating a flow is almost identical, but with a change in the userData and flowRate parameters:

```
//update a flow
async function main() {

  const web3 = new Web3(new Web3.providers.HttpProvider(process.env.MUMBAI_ALCHEMY_URL)
```

```
//create contract instances for each of these
const host = new web3.eth.Contract(hostABI, hostAddress);
const cfa = new web3.eth.Contract(cfaABI, cfaAddress);
const tradeableCashflow = new web3.eth.Contract(tradeableCashflowABI, tradeableCashfl

const fDAIx = "0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f"
const userData = web3.eth.abi.encodeParameter('string', 'HODL ETH');


const nonce = await web3.eth.getTransactionCount(_sender, 'latest'); // nonce starts

//create flow by calling host directly in this function
//create flow from sender to tradeable cashflow address
//pass in userData to the flow as a parameter
async function updateFlow() {
    let cfaTx = (await cfa.methods
   .updateFlow(
    fDAIx,
    // _sender,
    tradeableCashflowAddress,
    "6858024691358",
    "0x"
   )
   .encodeABI())

   let txData = (await host.methods.callAgreement(
```

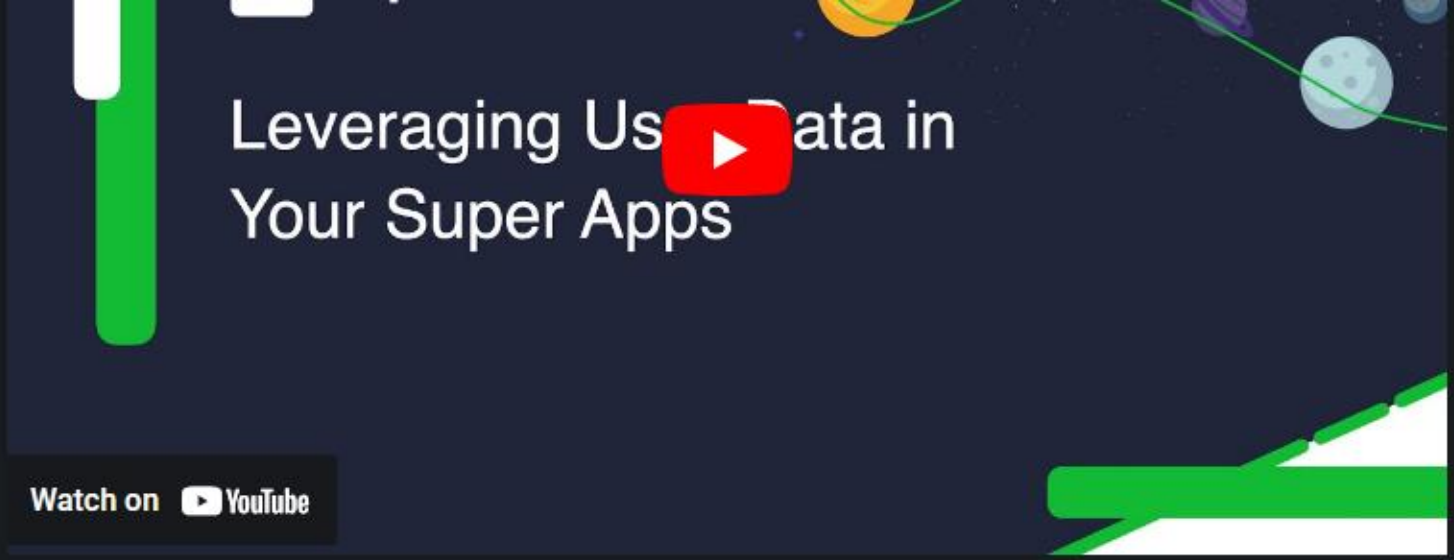
```
    let txData = (await hostMethodsDeclaration(
      cfaAddress,
      cfaTx,
      userData
    ).encodeABI());

    let tx = {
      'to': hostAddress,
      'gas': 3000000,
      'nonce': nonce,
      'data': txData
    }

    let signedTx = await web3.eth.accounts.signTransaction(tx, process.env.MUMBAI_DEPLO

    await web3.eth.sendSignedTransaction(signedTx.rawTransaction, function(error, hash)
      if (!error) {
        console.log("🎉 The hash of your transaction is: ", hash, "\n Check Alchemy's M
      } else {
        console.log("❗Something went wrong while submitting your transaction:", error)
      }
     });

    }

  await updateFlow();

  }
```

```
// We recommend this pattern to be able to use async/await everywhere
// and properly handle errors.
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

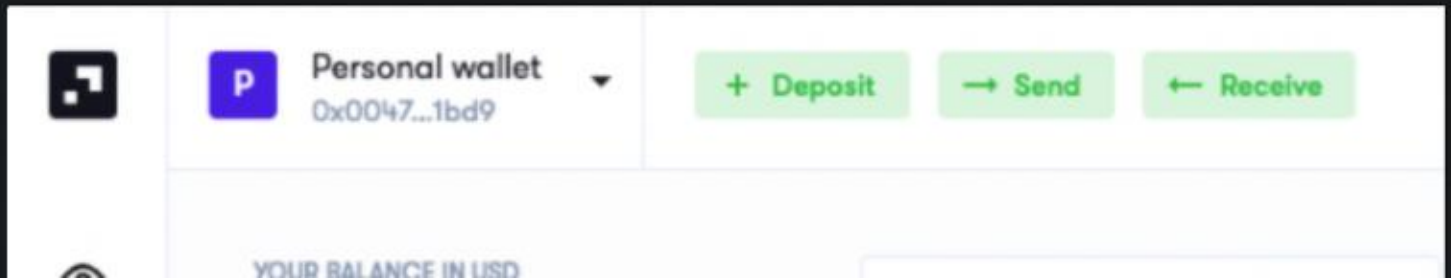
Our delete flow script has 2 key differences: there is no flowrate passed in, and no userData either. We set the value of userData inside of our contract back to an empty string when an agreement is terminated.

```
//delete a flow
async function main() {

const web3 = new Web3(new Web3.providers.HttpProvider(process.env.MUMBAI_ALCHEMY_URL));

  //create contract instances for each of these
  const host = new web3.eth.Contract(hostABI, hostAddress);
  const cfa = new web3.eth.Contract(cfaABI, cfaAddress);
  const tradeableCashflow = new web3.eth.Contract(tradeableCashflowABI, tradeableCashfl

  const _sender = "0x9421FE8eCcAfad76C3A9Ec8f9779fAfA05A836B3"
```

```
const accts = await web3.eth.getAccounts();

const fDAIx = "0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f"

const nonce = await web3.eth.getTransactionCount(_sender, 'latest'); // nonce starts

async function cancelFlow() {
    let cfaTx = (await cfa.methods
   .deleteFlow(
    fDAIx,
    _sender,
    tradeableCashflowAddress,
    "0x"
   )
   .encodeABI())
  //try using callAgreement vs callagreement w context
   let txData = (await host.methods.callAgreement(
    cfaAddress,
    cfaTx,
    //pass in empty field for userData
    "0x"
  ).encodeABI());

  let tx = {
    'to': hostAddress,
    'gas': 3000000,
```



Before we get started with project setup, you'll also want to head over to the Superfluid Dashboard to claim testnet Super DAI (fDAIx) in at least one account on the network you'd like to use for the tutorial (I would suggest claiming these tokens on Mumbai, the Matic Testnet). To claim test DAI, you can head to the currencies tab in the dashboard and click the plus button on the far right in the DAI row to get your hands on some test Super DAIx.



```
      'nonce': nonce,
      'data': txData
    }

    let signedTx = await web3.eth.accounts.signTransaction(tx, process.env.MUMBAI_DEPLO

    await web3.eth.sendSignedTransaction(signedTx.rawTransaction, function(error, hash)
      if (!error) {
        console.log("🎉 The hash of your transaction is: ", hash, "\n Check Alchemy's M
      } else {
        console.log("❗Something went wrong while submitting your transaction:", error)
      }
     });

    }


  await cancelFlow();

}

// We recommend this pattern to be able to use async/await everywhere
// and properly handle errors.
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
```

```
  process.exit(1);
});
```

Finally, we'll also create a script which helps us read on chain data that corresponds to our flow. This will allow us to see what Context looks like when passed around within the protocol, and it allows us to see what userData looks like when appended onto it as a bytes value. Once we can identify its place in the Context struct, we just need to decode it to make use of it.

```
//read flowData
async function main() {

const web3 = new Web3(new Web3.providers.HttpProvider(process.env.MUMBAI_ALCHEMY_URL));

  //create contract instances for each of these
  const host = new web3.eth.Contract(hostABI, hostAddress);
  const cfa = new web3.eth.Contract(cfaABI, cfaAddress);
  const tradeableCashflow = new web3.eth.Contract(tradeableCashflowABI, tradeableCashfl
  const fDAIx = "0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f"


  //get data
  const decodedContext = await tradeableCashflow.methods.uData().call();
  const decodedUserData = web3.eth.abi.decodeParameter('string', decodedContext.userDat
  console.log(decodedContext)
  console.log(decodedUserData)
```

```
  //get jail info
  const jailed = await host.methods.getAppManifest(tradeableCashflowAddress).call()
  console.log(jailed)
  const isJailed = await host.methods.isAppJailed(tradeableCashflowAddress).call();
  console.log(`is jailed: ${isJailed}`);

  const flowInfo = await cfa.methods.getFlow(fDAIx, tradeableCashflowAddress, "0x00471E
  const outFlowRate = Number(flowInfo.flowRate);
  console.log(`Outflow Rate: ${outFlowRate}`);

  const netFlow = await cfa.methods.getNetFlow(fDAIx, tradeableCashflowAddress).call();
  console.log(`Net flow: ${netFlow}`);

  const inFlowRate = Number(netFlow) + outFlowRate;
  console.log(`Inflow rate: ${inFlowRate}`)
}

// We recommend this pattern to be able to use async/await everywhere
// and properly handle errors.
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

A sample output from readData.js:

```
//decoded context in terminal

appLevel: '1',
callType: '3',
timestamp: '1634263831',
msgSender: '0x9421FE8eCcAfad76C3A9Ec8f9779fAfA05A836B3',
agreementSelector: '0x62fc305e',
userData: '0x0000000000000000000000000000000000000000000000000000000000000200000000000
appAllowanceGranted: '13888889148145664',
appAllowanceWanted: '0',
appAllowanceUsed: '0',
appAddress: '0xB04FCcCc7Ef4cfCff6F1693492DdE15BcEbEe71f',
appAllowanceToken: '0x5D8B4C2554aeB7e86F387B4d6c00Ac33499Ed01f'
}
//decoded userData
"HODL BTC"
```

If you want, you can also check out the Superfluid dashboard to see these flows being created in real time (you just can't see the userData there)!

The Front End - Displaying our Billboard

In this repository, you'll see that you have many of the other components and UI elements from scaffold-eth still there. We left them there to give you the opportunity to continue tinkering with the rest of the framework if you choose to. However, to display our NFT Billboard, we'll need to make a few changes.

First, within the views folder, we'll make a copy of the 'ExampleUI.jsx' file called 'NFTBillboard.jsx' and make a few changes.

We'll add `message` and `billboardOwner` to our list of exports, and remove some of the items from ExampleUI:

```
export default function NFTBillboard({
  message,
  billboardOwner,
  mainnetProvider,
  readContracts,
})
```

And we'll also change the content our function component returns to reflect our data:

```
return (
    <div>
      {/*
        ⚙ Here is a UI that displays and sets the message in your NFT Billboard:
      */}
      <div style={{ border: "1px solid #cccccc", padding: 16, width: 400, margin: "auto'
```

```
        <h1>NFT Billboard</h1>
        <h2>Message: <b>{message}</b></h2>
        <Divider />
        <div style={{ margin: 8 }}>
            <h3>Owner:</h3>
            <h4>
                <Address address={billboardOwner} />
            </h4>
        </div>

      </div>


      <div style={{ width: 600, margin: "auto", marginTop: 32, paddingBottom: 256 }}>

      Billboard Contract Address:
        <Address
          address={readContracts && readContracts.TradeableCashflow ? readContracts.Trac
          ensProvider={mainnetProvider}
          fontSize={16}
        />

      </div>
    </div>
```

If you peer into the original 'ExampleUI.jsx' file, you'll see that we removed a great deal of content.

All that's left is the top section and an Address component where we'll display our content's address. Again - you're welcome to utilize the other front end content if you'd like in your own applications 😎

Next, we need to add our NFTBillboard component to our App.jsx file, and make sure that we have logic that will read the message (i.e. `userData`) and owner of our billboard (the `_receiver` we pass in when we deployed our contract).

At the top of our file, we'll import the component

```
//in the top section of our App.jsx file
//add NFT billboard here
import NFTBillboard from "./views/NFTBillboard";
```

We'll then need to make sure that our react app is reading data from the right network. To do this, we'll need to make a change to targetNetwork . In our case, we'll set the network to NETWORKS.mumbai - but if you're using a different network, you'll need to specify that here.

```
//on line 57 within App.jsx
/// 📡 What chain are your contracts deployed to?
const targetNetwork = NETWORKS.mumbai; // <------- select your target frontend network
```

To get the values for the billboard owner and message, we'll need to use the useContractReader hook that's included in scaffold-eth from eth-hooks. You can read more about eth-hooks here.

The scaffold-eth framework already defines a helpful `readContracts` variable that we can pass to useContractReader which will allow us to read data from our deployed contracts. If you want to create new contracts that you can write to, you'll use the `writeContracts` variable instead.

```
//already included for us from Scaffold-eth
// Load in your local 📝 contract and read a value from it:
  const readContracts = useContractLoader(localProvider, contractConfig);
```

We can set our message and billboardOwner variables like this:

```
//two new variables that we'll use
const message = useContractReader(readContracts, "TradeableCashflow", "userData")
const billboardOwner = useContractReader(readContracts, "TradeableCashflow", "_receiver'
```

Finally, we'll include our NFTBillboard component in our body, and set the "/" route to render our billboard component.

```
// Some code
//at the top of our Switch statement
<Switch>
<Route exact path="/">

<NFTBillboard
```

```
<NFTBillboard
    address={address}
    mainnetProvider={mainnetProvider}
    readContracts={readContracts}
    billboardOwner={billboardOwner}
    message={message}

  />

//...
</Route>
</Switch>
```

You'll notice that much of the body in App.jsx is commented out. If you're curious about what the other elements do, feel free to uncomment them and play around.

There you have it! You can run `yarn start` to run your react app locally, then `yarn deploy` on your contracts to see things render! To send your message, you can call `createFlow` to see the message appear on your billboard in real time.

I would also recommend opening up the superfluid dashboard to see how flows are moving into and out of your NFT contract. In the account you're using to deploy the contract, you'll see that you have a flow created into the contract.

Then, if you look at the account you used for the owner of the tradeable cashflow (i.e. billboard) contract, you'll see that you have an incoming stream from the contract itself.

```
//...
</Route>
</Switch>
```

You'll notice that much of the body in App.jsx is commented out. If you're curious about what the other elements do, feel free to uncomment them and play around.

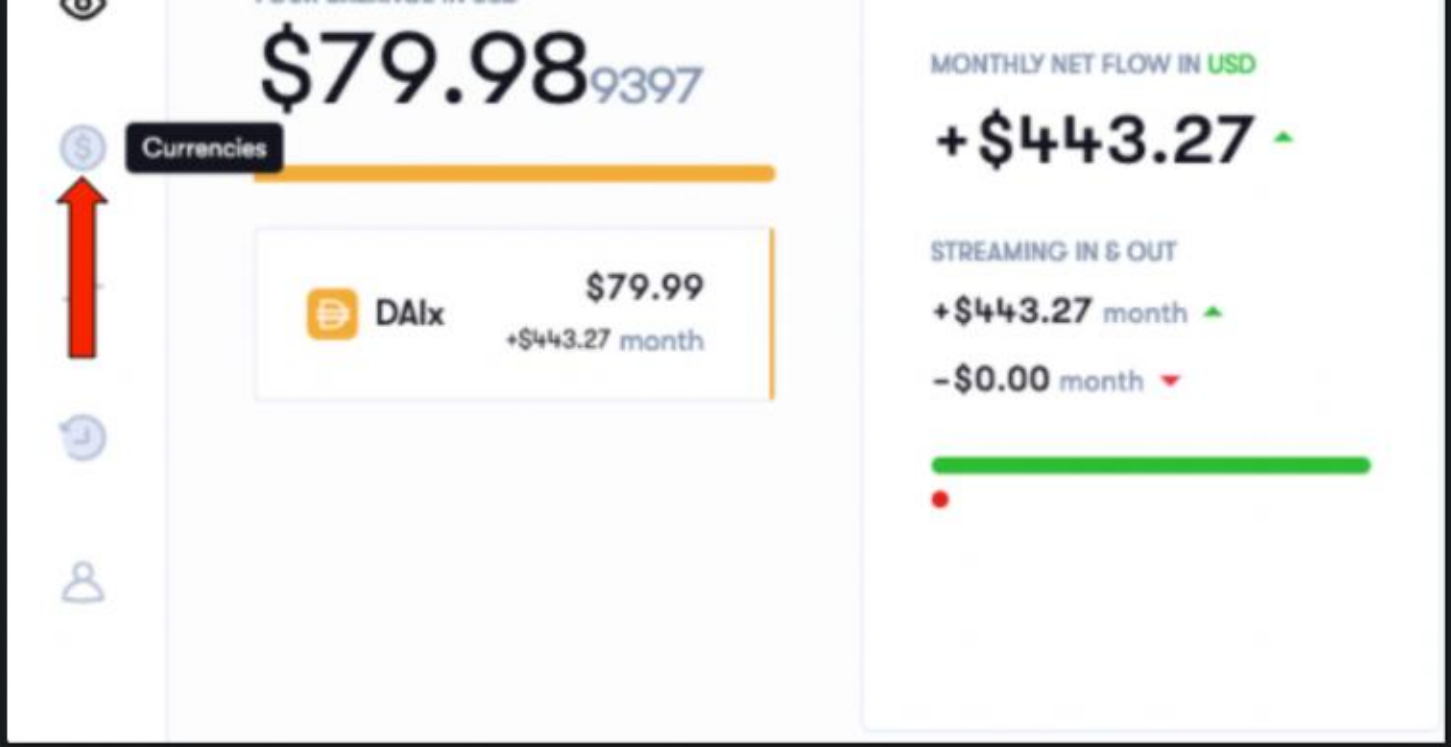
There you have it! You can run `yarn start` to run your react app locally, then `yarn deploy` on your contracts to see things render! To send your message, you can call `createFlow` to see the message appear on your billboard in real time.

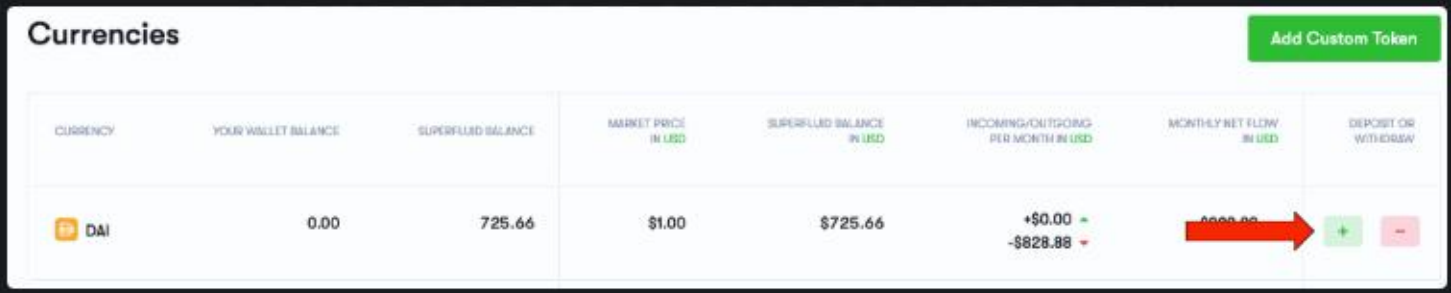
I would also recommend opening up the superfluid dashboard to see how flows are moving into and out of your NFT contract. In the account you're using to deploy the contract, you'll see that you have a flow created into the contract.

Then, if you look at the account you used for the owner of the tradeable cashflow (i.e. billboard) contract, you'll see that you have an incoming stream from the contract itself.

← Previous
User Data

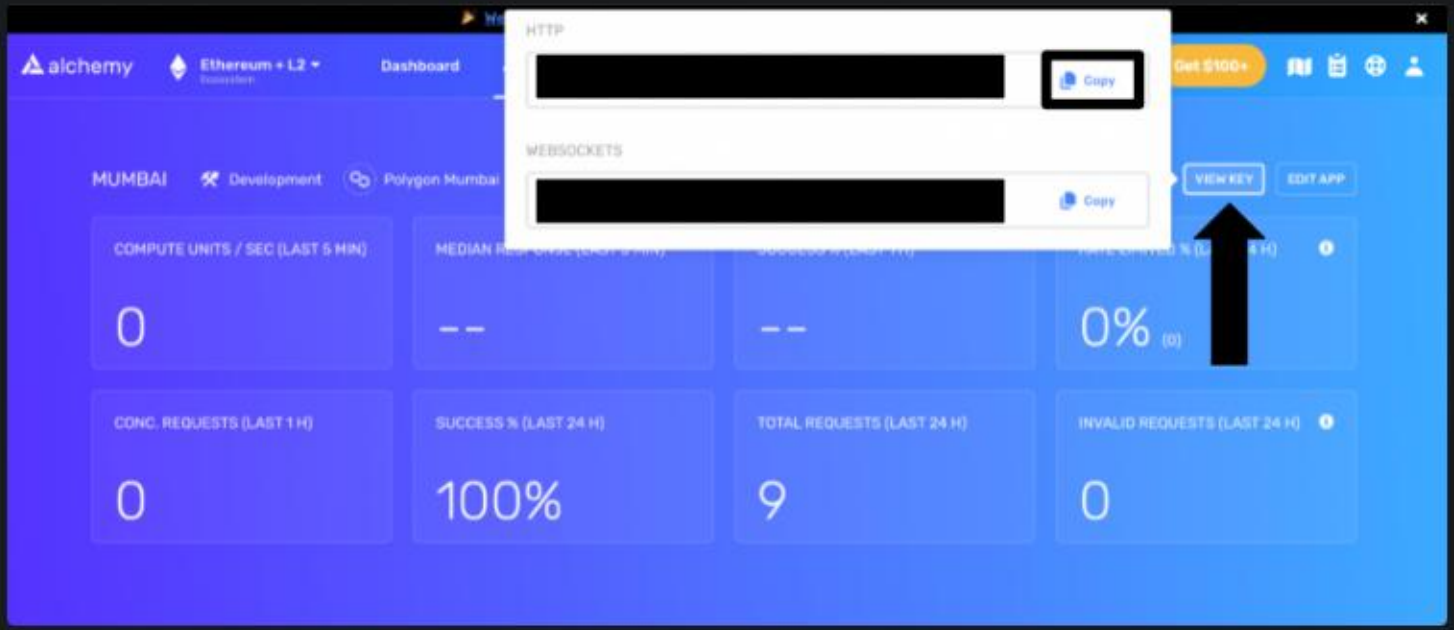
Next
SuperAppBaseFlow →

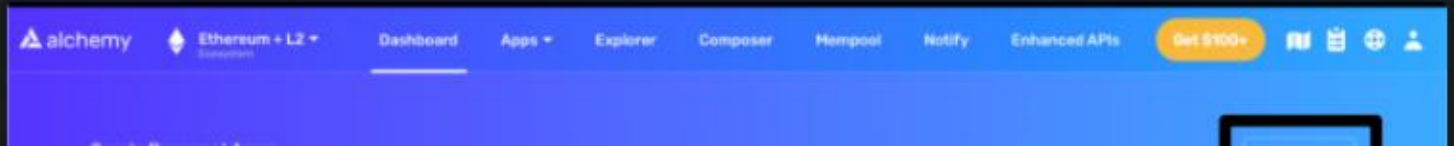

$79.98 9397
Currencies
DAIx
$79.99
+$443.27 month
MONTHLY NET FLOW IN USD
+$443.27
STREAMING IN & OUT
+$443.27 month
-$0.00 month

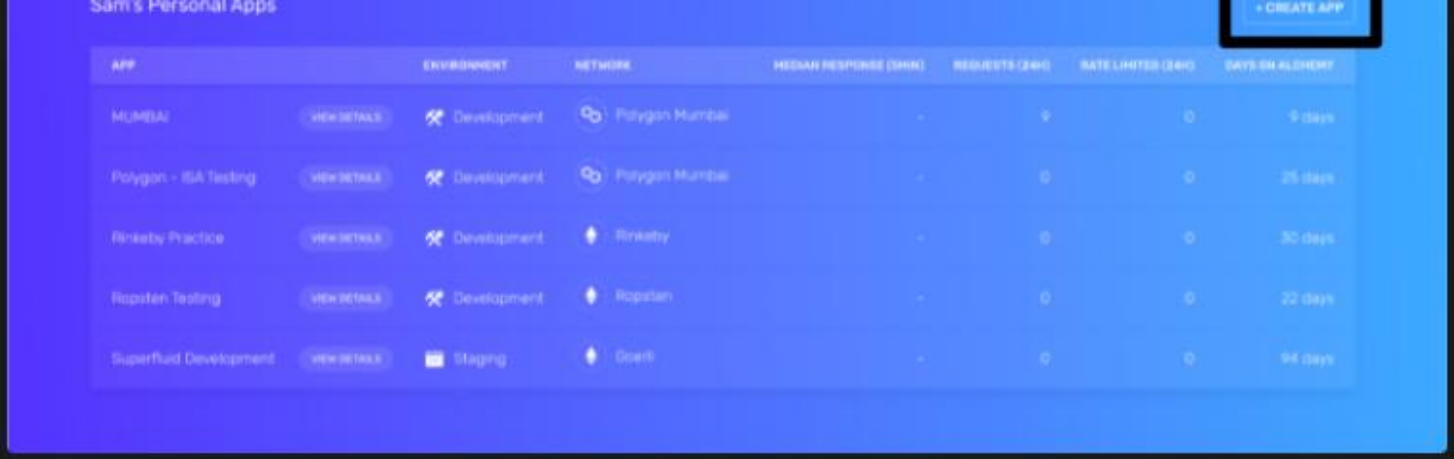

Currencies
Add Custom Token
CURRENCY
YOUR WALLET BALANCE
SUPERFLUID BALANCE
MARKET PRICE IN USD
SUPERFLUID BALANCE IN USD
INCOMING/OUTGOING PER MONTH IN USD
MONTHLY NET FLOW IN USD
DEPOSIT OR WITHDRAW
DAI
0.00
725.66
$1.00
$725.66
+$0.00
-$828.88

You'll want to have 2 Ethereum addresses ready: one of which you'll need your private key for (that has our test tokens), the other which we'll just be observing.

Scaffold-Eth and Hardhat Configuration







To get your URL, click on your project, select 'View Key' and copy the HTTP URL.

You'll want to create a new .env file and put this URL there - in my case I've called it `MUMBAI_ALCHEMY_URL`.

I've also gotten the private key associated with the address I'll be using to deploy contracts and create flows. I've set this as an environment variable as well called `MUMBAI_DEPLOYER_PRIV_KEY`.

> NOTE: be VERY careful with your private keys. Do not push them to github or share them publicly. If you need help locating your keys within metamask, you can click on 'Account Details' in the menu and enter your password to 'Export Private Key.'

The standard hardhat.config file will have quite a few network options to choose from. However, in our case, I need to change my default network to "polytest," and include my Alchemy URL and keys under 'accounts' so that hardhat can use my account as a signer when deploying contracts.

Note: if you choose to use a different testnet, you'll need to change the default network to network you wish to use.

```
//inside of hardhat.config on line 29

// Select the network you want to deploy to here:
//
//using polytest (mumbai)
const defaultNetwork = "polytest";
```

Next, scroll down to 'polytest' in the list of potential networks that Scaffold-eth lists in hardhat.config. Add in your URL from Alchemy in 'url' and your private key in 'accounts.' Note: you'll need to prepend '0x' onto your private key. Again - these should be saved as environment variables - don't openly share your private key!

Note: if you choose to use a different testnet, you'll need to add this data for the network you wish to use (i.e. goerli, ropsten, etc.)

```
//inside of hardhat.config on line 163
polytest: {
    url: `${process.env.MUMBAI_ALCHEMY_URL}`,// using alchemy instead of moralis. add y
    gasPrice: 1000000000,
    accounts: [`0x${process.env.MUMBAI_DEPLOYER_PRIV_KEY}`]
    },
 },
```

Finally, we need to adjust our solidity compiler so that it's compatible with our project.

```
//in hardhat.config on line 280
solidity: {
    compilers: [
      {
                        //set compiler to version 0.7.0
        version: "0.7.0",
        settings: {
          optimizer: {
            enabled: true,
            runs: 200,
          },
        },
      },
      {
        version: "0.6.7",
        settings: {
          optimizer: {
            enabled: true,
            runs: 200,
          },
        },
      },
```

```
    ],
  }
```

Contracts

Next, we'll add in our two key contracts: RedirectAll.sol and TradeableCashflow.sol. As I mentioned, these are very similar to the example in our SuperApps tutorial, but with a few small changes to the RedirectAll.sol contract:

1) We need to add variables which will store the incoming context and userData within the callbacks

```
//public variables which we'll set userData values to
    ISuperfluid.Context public uData;
    string public userData;
```

2) We'll add new logic at the bottom of each callback:

Inside of the `afterAgreementCreated` callback:

```
// decode Context - store full context as uData variable for easy visualization purpose:
ISuperfluid.Context memory decompiledContext = _host.decodeCtx(_ctx);
uData = decompiledContext;

//set userData variable to decoded value
//for now, this value is hardcoded as a string - this will be made clear in flow creati
```

```
//this string will serve as a message on an 'NFT billboard' when a flow is created with
//it will be displayed on a front end for assistance in userData explanation

userData = abi.decode(decompiledContext.userData, (string));
```

Inside of the `afterAgreementUpdated` callback:

```
//update the context with the same logic...

ISuperfluid.Context memory decodedContext = _host.decodeCtx(_ctx);
uData = decodedContext;
userData = abi.decode(decodedContext.userData, (string));
```

Inside of the `afterAgreementTerminated` callback:

```
//set the userData (i.e. the billboard message) to an empty string
userData = "";
```

This logic will take in the message passed in as userData to calls made to the constant flow agreement which target the flow into this app, decode it, and set it to a storage variable so that we can very easily see what's going on throughout this process. Feel free to make this code your own and do your own gas optimization as you see fit 😁

SuperAppBaseFlow

Making building Super Apps easier than ever

> You may see the name "SuperAppBaseCFA" instead of SuperAppBaseFlow around our super-examples repo. We are in the process or renaming; eventually all mentions of SuperAppBaseCFA will be replaced with SuperAppBaseFlow.

What is the SuperAppBaseFlow

The SuperAppBaseFlow is an inheritable base contract that simplifies writing callbacks and abstracts away the redundancies of writing Super Apps. It can be used in lieu of the SuperAppBase.

SuperAppBaseFlow Code

protocol-monorepo/SuperAppBaseFlow.sol at dev · superfluid-finance/protocol-mono...
GitHub

Example Usage - Growing NFT Example Using SuperAppBaseFlow

super-examples/Flower.sol at c784d239557d6fb5e56a2c8951ac4353256d611d · super...
GitHub

SuperAppBaseFlow Simplifies Callbacks

The original *SuperAppBase* callback function headers are elaborate with unobvious parameter meanings. The SuperAppBase*Flow* consolidates all callback development into 3 virtual functions (`onFlowCreated` , `onFlowUpdated` , `onFlowDeleted`) with much more readable parameters, making development easier for you.

SuperAppBase

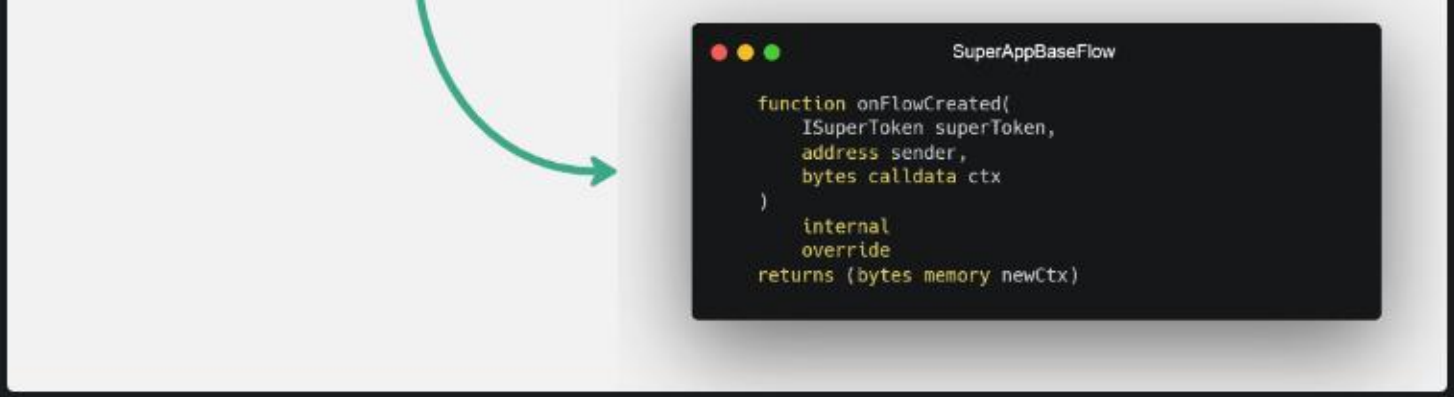
```
function afterAgreementCreated(
    ISuperToken superToken,
    address agreementClass,
    bytes32 agreementId,
    bytes calldata agreementData,
    bytes calldata cbdata,
    bytes calldata ctx
) external override returns (bytes memory newCtx) {
```

	the Super App was last modified.
`ctx`	Contains the context of the stream being modified to the Super App. The context gets updated when Super Agreement modifications are done in the callback body (learn more). The context must be returned at the end of the callback.

* If a stream is being deleted by the account streaming to the Super App, `sender` is the streaming account and `receiver` is the Super App (obviously). However, if the Super App itself is deleting a flow it's receiving, then that triggers `onFlowDeleted` as well. In this case, it's flipped - `sender` is the Super App and `receiver` is the streaming account.

← Previous
NFT Billboard Example

Next
Super App Examples →



Example: onFlowCreated is a more intuitive function than afterAgreementCreated

Using the SuperAppBaseFlow

Import and Inherit

```
// Example Code
...
import { SuperAppBaseCFA } from "@superfluid-finance/ethereum-contracts/contracts/apps/

contract SomeSuperAppContract is SuperAppBaseCFA {
...
```

Constructor

```
constructor(
    ISuperfluid host_,
    bool activateOnCreated,
    bool activateOnUpdated,
    bool activateOnDeleted
)
```

`host_` - The address of the Superfluid Host for the network of choice. Find addresses here.

`activateOnCreated` - True if the Super App will implement `onFlowCreated` to react to the creation of streams to it. False otherwise.

`activateOnUpdated` - True if the Super App will implement `onFlowUpdated` to react to the updating of streams being sent to it. False otherwise.

`activateOnDeleted` - True if the Super App will implement `onFlowDeleted` to react to the deletion of streams being sent to it. False otherwise.

Token Acceptance

The SuperAppBaseFlow exposes a virtual `isAcceptedSuperToken` function, which can be overridden to specify streams of which Super Tokens the Super App's callbacks can react to.

You can forgo it and just use your own functions/modifiers/etc., but using `isAcceptedSuperToken` is the simplest and most convenient way to set up Super Token

`isAcceptedSuperToken` is the simplest and most convenient way to set up Super Token acceptance.

```
/**
 * @dev Optional (positive) filter for accepting only specific SuperTokens.
 *      The default implementation accepts all SuperTokens.
 *      Can be overridden by the SuperApp in order to apply arbitrary filters.
 */
function isAcceptedSuperToken(ISuperToken /*superToken*/) public view virtual returns
    return true;
}
```

> ⓘ Every Super App ought to have functionality to restrict which Super Tokens can trigger callbacks to protect against vulnerabilities from malicious custom Super Tokens.

Callback Functions

`onFlowCreated`

Override with custom logic that will trigger when a new flow to the Super App is **created**.

Example from Tradeable Cashflow Example Project

```
/// @dev override if the SuperApp shall have custom logic invoked when a new flow
///      to it is created.
function onFlowCreated(
    ISuperToken superToken,
    address sender,
    bytes calldata ctx
) internal virtual returns (bytes memory /*newCtx*/) {
    return ctx;
}
```

Parameter	Description
`superToken`	Super Token being streamed to the Super App
`sender`	Address streaming `superToken` to the Super App
`ctx`	Contains the context of the stream being modified to the Super App. The context gets updated when Super Agreement modifications are done in the callback body (learn more).

	The context must be returned at the end of the callback.

`onFlowUpdated`

Override with custom logic that will trigger when an existing flow to the Super App is **updated**.

Example from Growing NFT Example Project

```
/// @dev override if the SuperApp shall have custom logic invoked when an existing flow
///      to it is updated (flowrate change).
function onFlowUpdated(
    ISuperToken superToken,
    address sender,
    int96 previousFlowRate,
    uint256 lastUpdated,
    bytes calldata ctx
) internal virtual returns (bytes memory /*newCtx*/) {
    return ctx;
}
```

Parameter	Description
	SuperToken being streamed to the Super

`superToken`	Super Token being streamed to the Super App
`sender`	Address streaming `superToken` to the Super App
`previousFlowRate`	The flow rate of the sender's stream to the Super App before the latest update.
`lastUpdated`	The timestamp of when a sender's stream to the Super App was last modified.
`ctx`	Contains the context of the stream being modified to the Super App. The context gets updated when Super Agreement modifications are done in the callback body (learn more). The context must be returned at the end of the callback.

`onFlowDeleted`

Override with custom logic that will trigger when an existing flow to the Super App is **deleted**.

Example from Borrow Against Salary Example Project

> ⚠ If `onFlowDeleted` ever reverts, your Super App will get "jailed" causing its callbacks to no longer trigger in response to stream modifications. So, write your `onFlowDeleted` code such that it won't revert! Be careful with your logic and think about using try/catches where necessary.

```
/// @dev override if the SuperApp shall have custom logic invoked when an existing flow
///      to it is deleted (flow rate set to 0).
///      Unlike the other callbacks, this method is NOT allowed to revert.
///      Failing to satisfy that requirement leads to jailing (defunct SuperApp).
function onFlowDeleted(
    ISuperToken superToken,
    address sender,
    address receiver,
    int96 previousFlowRate,
    uint256 lastUpdated,
    bytes calldata ctx
) internal virtual returns (bytes memory /*newCtx*/) {
    return ctx;
}
```

Parameter	Description
`superToken`	Super Token being streamed to the Super App
`sender`	Address of the account streaming `superToken` to the Super App (or address of the account streaming `superToken` to the Super App if the Super App is deleting its own flow) *
`receiver`	Address of Super App (or address of Super App if Super App is deleting its own flow) *
`previousFlowRate`	The flow rate of the sender's stream to the Super App before the latest update.
`lastUpdated`	The timestamp of when a sender's stream to the Super App was last modified.
`ctx`	Contains the context of the stream being modified to the Super App. The context gets updated when Super Agreement modifications are done in the callback body (learn more).

Types of Super Tokens

Wrapper, Pure, Native

1. Wrapper Super Tokens

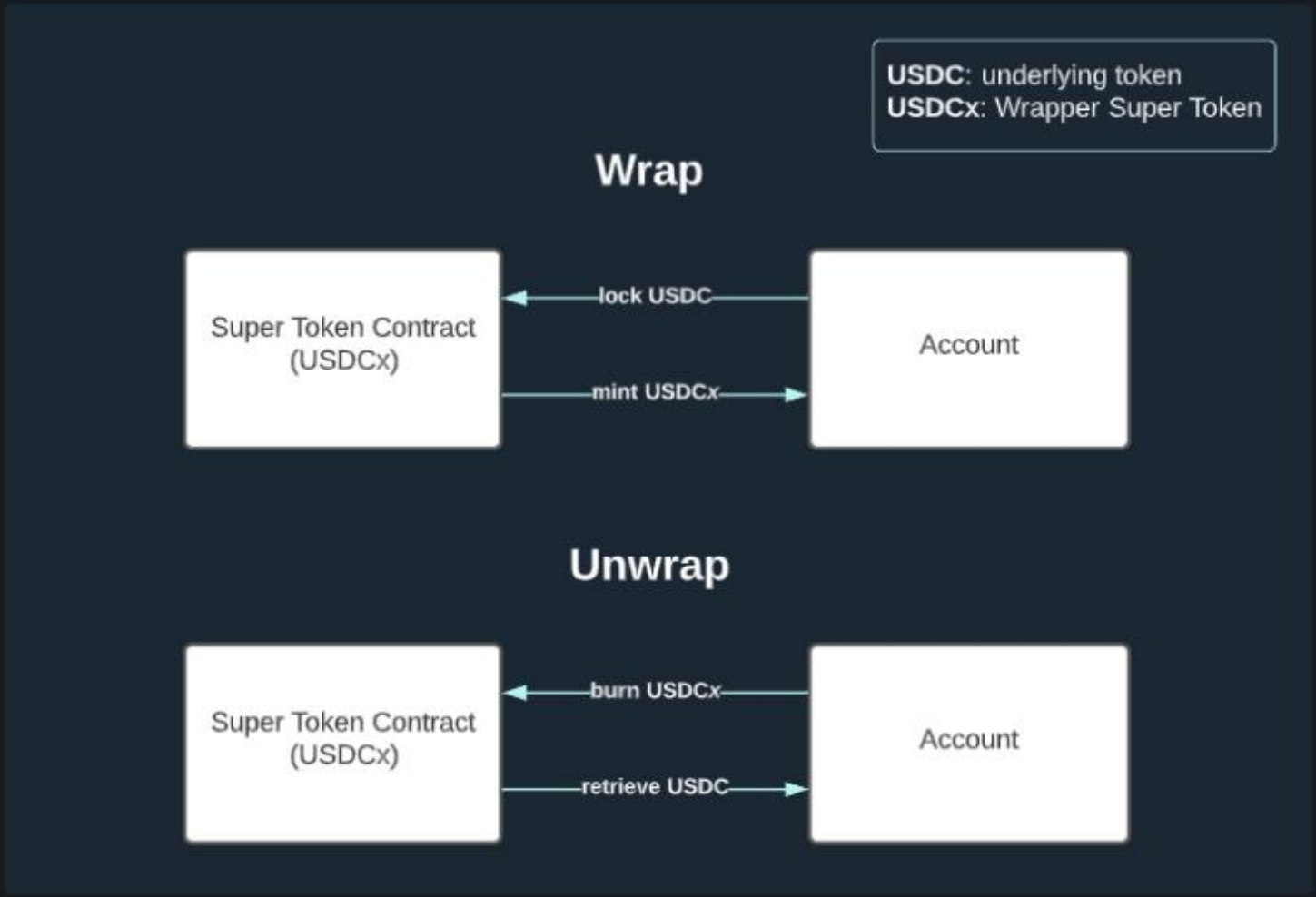
Wrapper Super Tokens are the most common Super Tokens. They are deployed whenever an ERC20 token already exists to create an equivalent Super Token version that has Super Agreement capabilities - like money streaming!

Anyone can permissionlessly create a wrapper for any existing ERC20 token. Wrappers for many popular tokens like DAI, USDC, and WETH have been deployed on the various chains we support. Check out the Network Directory for the full list.

Wrapping & Unwrapping

When getting wrapper Super Tokens, you first deposit the underlying tokens into its Wrapper Super Token contract after which you are minted an equal quantity of wrapper Super Tokens. Basically, you gave up the plain underlying token and got the "Super Token version" of it.

To unwrap, the opposite happens; a desired amount of Super Tokens are burned and an equal amount of the underlying token is returned.



Custom Wrapper Super Tokens

Custom Wrapper Super Tokens incorporate additional logic on the wrapper contract beyond the standard wrap/unwrap. They typically are used when the underlying ERC20 token has advanced logic that would get stifled by a standard Wrapper Super Token.

The only example of this would be the StIbAlluo token by Alluo which serves as a Custom Wrapper Super Token for the complex interest-bearing IbAlluo token. Designing a Custom Wrapper is a different challenge case-by-case, so if you decide it is best for you, please let us know over #dev-support so we can assist you.

2. Pure Super Tokens

Pure Super Tokens don't have an underlying token. They exist only as Super Tokens so there's no wrapping or unwrapping required. Pure Super Tokens are deployed directly as Super Tokens with all of the associated Superfluid functionality out of the box. These tokens inherently have all the functionality of a basic ERC20 token plus Superfluid's Super Agreement abilities.

Customization

You may want to give your Pure Super Tokens other kinds of non-Superfluid functionalities such as...

- Pre-mint initial supply
- Access control and admin management
- Wallet whitelisting

- ...the list goes on

All this logic can be programmed into your Pure Super Token just as you would add your own custom logic to an ERC20.

Governed vs Independent

"Governed" Pure Super Tokens are connected with Superfluid Protocol Governance such that when upgrades are implemented, they immediately reflect the new logic. **We encourage Pure Super Tokens be deployed as governed**.

"Independent" Pure Super Tokens are fully separate from Superfluid Protocol Governance (you can think of them as "jailbroken" Super Tokens). All upgrades are managed solely by the token developers. This type is not recommended it but if you do decide it is best for you, please let us know over #dev-support so we can assist you.

3. Native Super Tokens

A native asset super token is similar to an Wrapper Super Token in that the contract itself is a wrapper around an existing asset. Native Super Token contracts allow you to deposit a native asset, such as ETH, MATIC, and AVAX into the contract, and receive a corresponding amount of the wrapped native asset in return (e.g. MATICx for MATIC).

Native Super Tokens have canonical deployments for each chain Superfluid deploys on. You can find the addresses in Contract Addresses.

Tokens be deployed as governed.

"Independent" Pure Super Tokens are fully separate from Superfluid Protocol Governance (you can think of them as "jailbroken" Super Tokens). All upgrades are managed solely by the token developers. This type is not recommended it but if you do decide it is best for you, please let us know over #dev-support so we can assist you.

3. Native Super Tokens

A native asset super token is similar to an Wrapper Super Token in that the contract itself is a wrapper around an existing asset. Native Super Token contracts allow you to deposit a native asset, such as ETH, MATIC, and AVAX into the contract, and receive a corresponding amount of the wrapped native asset in return (e.g. MATICx for MATIC).

Native Super Tokens have canonical deployments for each chain Superfluid deploys on. You can find the addresses in Contract Addresses.

← Previous
Super Tokens - Frontend Examples

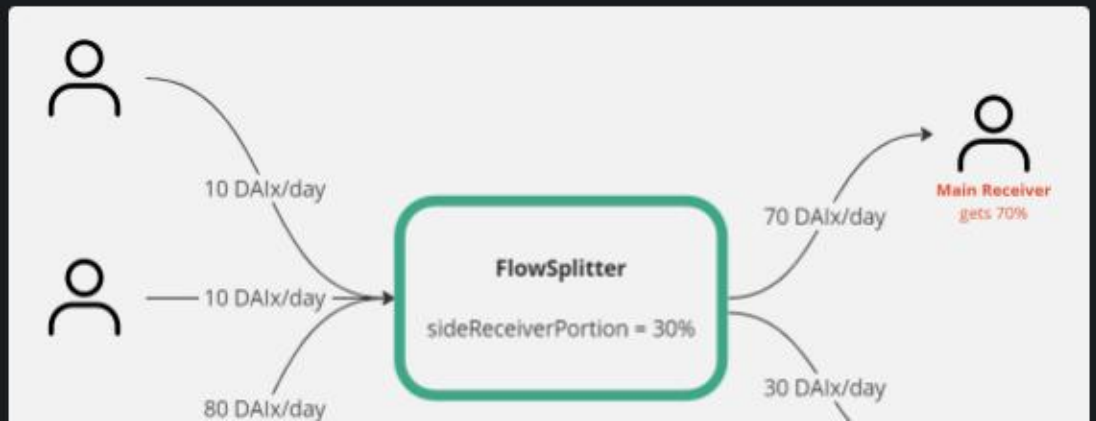
Next
Deploy a Super Token →

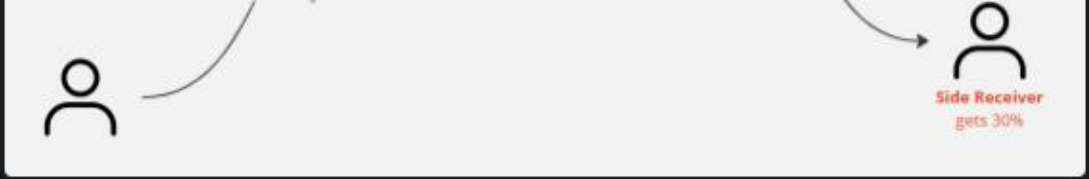
Super App Examples

Flow Splitter 🪓

super-examples/projects/flow-splitter at main · superfluid-finance/super-examples
GitHub

The Flow Splitter is a Super App that reacts to an incoming money stream by splitting it into two outgoing money streams between two receiver accounts.

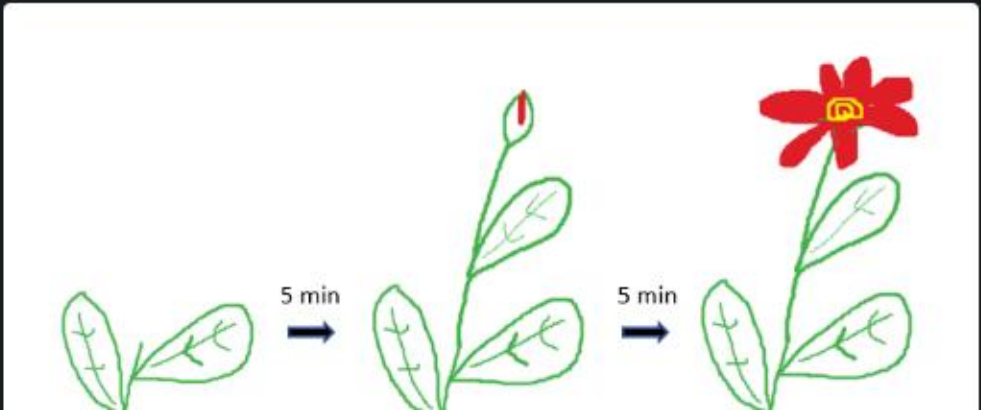




Growing NFT 🌼

super-examples/projects/growing-nft at main · superfluid-finance/super-examples
GitHub

The Growing NFT example demonstrates the Flower NFT Contract, an ERC721 contract that is also a Super App. As the amount of Super Tokens you stream to it crosses certain thresholds, the NFT's metadata will progressively graduate.



Tradeable Cashflow

super-examples/projects/tradeable-cashflow at main · superfluid-finance/super-exam...
GitHub

The Tradeable Cashflow provides the deployer with an NFT. Whoever is the owner of the NFT will receive a consolidated stream for all the streams the contract receives.

Previous
SuperAppBaseFlow

Next
Super App Deployment Guide

Last modified 1mo ago

WAS THIS PAGE HELPFUL?

Tradeable Cashflow

super-examples/projects/tradeable-cashflow at main · superfluid-finance/super-exam...
GitHub

The Tradeable Cashflow provides the deployer with an NFT. Whoever is the owner of the NFT will receive a consolidated stream for all the streams the contract receives.

Previous
SuperAppBaseFlow

Next
Super App Deployment Guide

Last modified 1mo ago

WAS THIS PAGE HELPFUL?

Super App Deployment Guide

Obtaining keys for super app deployment

Looking for instructions on how to deploy a super app? Check out our guide on github:

Super App White listing Guide · superfluid-finance/protocol-monorepo Wiki
GitHub

Check out this thorough guide on mainnet deployment and forking from omnifient, a community contributor.

Superfluid && Mainnet Fork

Previous
Super App Examples

Next - Developers
Batch Calls

Obtaining keys for super app deployment

Looking for instructions on how to deploy a super app? Check out our guide on github:

Super App White listing Guide · superfluid-finance/protocol-monorepo Wiki
GitHub

Check out this thorough guide on mainnet deployment and forking from omnifient, a community contributor.

Superfluid && Mainnet Fork

Previous
Super App Examples

Next - Developers
Batch Calls

Last modified 8mo ago

Batch Calls

The Batch Call feature allows you to perform multiple steps in a single transaction, including calls for Super Tokens, Super Apps, and both at the same time.

The primary goal of the Batch Call is to provide you (and your users) with an easy on/off-ramp to your Super App. It allows smooth user-experiences, without spending precious resources on solidity research and engineering for ad-hoc multi-step transactions.

Instead, you can focus on writing your Super App quickly and intuitively. Once you're satisfied with the basic functionality, it's only a matter of "wrapping up" functions into a Batch Call.

Previous
Super App Deployment Guide

Next
Batch Calls - SDK Core

Last modified 10mo ago

Batch Calls

The Batch Call feature allows you to perform multiple steps in a single transaction, including calls for Super Tokens, Super Apps, and both at the same time.

The primary goal of the Batch Call is to provide you (and your users) with an easy on/off-ramp to your Super App. It allows smooth user-experiences, without spending precious resources on solidity research and engineering for ad-hoc multi-step transactions.

Instead, you can focus on writing your Super App quickly and intuitively. Once you're satisfied with the basic functionality, it's only a matter of "wrapping up" functions into a Batch Call.

Previous
Super App Deployment Guide

Next
Batch Calls - SDK Core

Last modified 10mo ago

Batch Calls - SDK Core

With Superfluid, you can batch many, many operations into a single transaction. Here's how you do it with the SDK Core

The `BatchCall` class allows the user to batch multiple supported operations/transactions in one operation. Similar to the other helper classes, we can create this either through the `Framework` or directly initialize this.

Supported Operations

Not all operations are supported by the batch call feature, below is a list of the supported operations:

- `ERC20_APPROVE (SuperToken only)`
- `ERC20_TRANSFER_FROM`
- `SUPERTOKEN_UPGRADE`
- `SUPERTOKEN_DOWNGRADE`
- `SUPERFLUID_CALL_AGREEMENT`
- `CALL_APP_ACTION`

Most of the token methods are self explanatory, but some additional context for the last two operations is helpful. `SUPERFLUID_CALL_AGREEMENT` refers to all operations related to the CFA or IDA (`createFlow`, `updateIndex`, `distribute`, etc.). `CALL_APP_ACTION` refers to an operation which is created from calling a function that exists on a super app you have created. Refer to Usage below to see how you can create a `CALL_APP_ACTION` operation.

Framework based initialization

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const sf = await Framework.create({
  chainId: 137, //chain Id for matic - change depending on current network
  provider
});

const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const batchCall = sf.batchCall([<OPERATION_A>, <OPERATION_B>, ...]);
```

Direct Initialization

```
import { SuperToken } from "@superfluid-finance/sdk-core";

const batchCall = new BatchCall({
  hostAddress: "0x3E14dC1b13c488a8d5D310918780c983bD5982E7",
  operations: [<OPERATION_A>, <OPERATION_B>, ...],
});
```

Usage

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const sf = await Framework.create({
  chainId: 137, //matic chainId
  provider
});

//load the token you'd like to use like this
//note that tokens may be loaded by symbol or by address
//this is daix on rinkeby
```

```
const daix = await sf.loadSuperToken("0x745861AeD1EEe363b4AaA5F1994Be40b1e05Ff90");

 // 0xabc is the signer on Rinkeby testnet
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });
const fromAddress = "0xabc";
const paymentAddress = "0xdef";
const approveOp = daix.approve({ receiver: paymentAddress, amount: "10000" });
const transferFromOp = daix.transferFrom({
  sender: fromAddress,
  receiver: paymentAddress,
  amount: "10000",
});
const batchCall = sf.batchCall([approveOp, transferFromOp]);
const txn = await batchCall.exec(signer);

// creating an operation from a super app function with callAppAction
// initialize your super app contract's interface
const superAppInterface = new ethers.utils.Interface(<SUPER_APP_ABI>);

// get the calldata for your transaction
in this case we're calling a hypothetical function titled 'transferERC20'
const callData = appInterface.encodeFunctionData("transferERC20", [
   token,
   receiver,
   amount,
   "0x",
]);
```

```
// get the calldata for your transaction
in this case we're calling a hypothetical function titled 'transferERC20'
const callData = appInterface.encodeFunctionData("transferERC20", [
   token,
   receiver,
   amount,
   "0x",
]);

//create the operation by using sf.host.callAppAction
const transferOp = sf.host.callAppAction(appAddress, callData);

//add the operation to a batch call
const batchCall = sf.batchCall([transferOp]);
await batchCall.exec(signer);
```


Developers - Previous
Batch Calls

Next
Batch Calls - Frontend Examples

Batch Calls - Frontend Examples

Batching transactions to improve your UX

Creating a Batch Call

Another game changing feature of the Superfluid protocol is the ability to batch transactions together to improve the quality of UX for your users. We elaborate much more on batch calls inside of our guide on the topic. However, what you need to know for this interactive section is that batch calls can be created with the following operations:

Type	Text	Internal Call
Super Token	ISuperToken.sol	`operationApprove`, `operationTransferFrom`, `operationUpgrade`, `operationDowngrade`
Agreement	ISuperfluid.sol	`_callAgreement`

Super App	ISuperfluid.sol	`_callAppAction`

> ⓘ You can learn more about `callAppAction` in our sdk overview section - it will allow you to add any function on your Super App contract to a batch call - even if that function performs functionality that's outside of the normal scope of a superfluid operation (such as a standard ERC20 transfer or calling another contract of a separate DeFi protocol)

In our example, we'll be focusing on a simple usage of batch calls: upgrading tokens to super tokens, and creating a flow in the same transaction. We'll achieve this by using the `operationUpgrade` super token call, and the `_callAgreement` call (to create the flow).

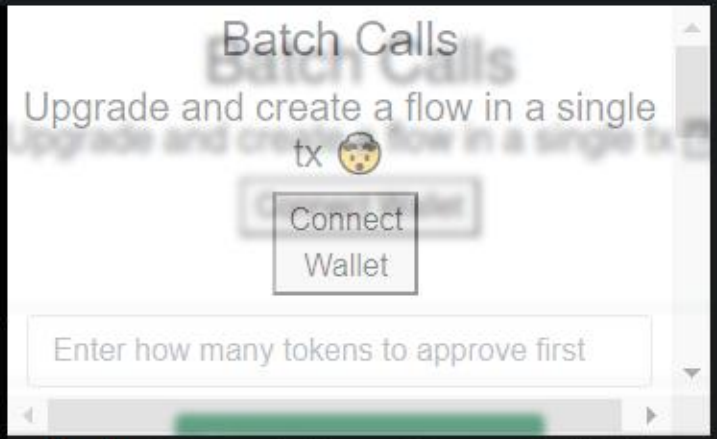
First, we'll need to approve the super token contract (in this case, fDAIx on Kovan) to spend our fDAI. Because this is an operation that is done outside of the Superfluid protocol on the ERC20 token contract, it cannot be batched.

After the approval, you can enter the number of tokens you'd like to upgrade, and information about the flow you'd like to create and click the 2nd button to run your batch call. You'll be able to see the upgrade and createFlow operation happen in a single transaction (just check the console for details!)

> ⓘ NOTE: we recommend using the Superfluid Console and the Superfluid Dashboard as you go through these interactive tutorials.

BatchCall.js

```
1  import React, { useState, useEffe
2  import { daiABI } from "./config"
3  import { Framework } from "@super
4  import {
5    Button,
6    Form,
7    FormGroup,
8    FormControl,
9    Spinner,
10   Card
11 } from "react-bootstrap";
12 import "./batchCall.css";
13 import { ethers } from "ethers";
14
15 let account;
```



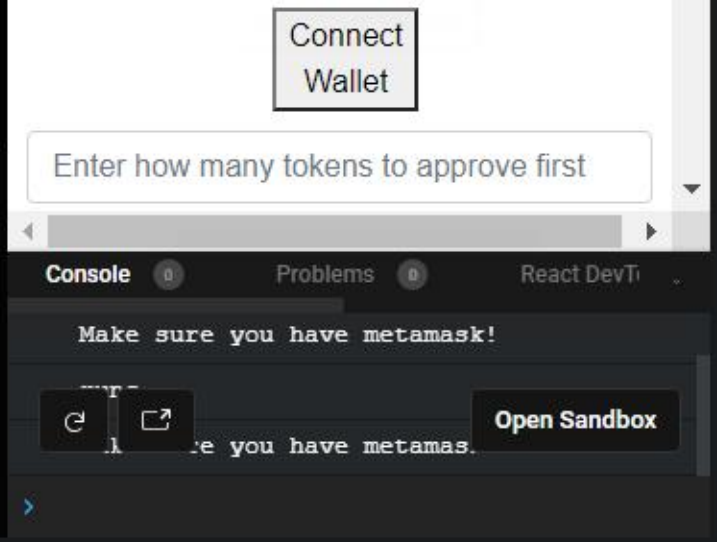
Approve the DAI spend, then enter an upgrade amount and the receiver + flowRate for your stream

Keep in mind that this is a short example, you could run many upgrade operations & callAgreement operations in this single transaction if you wanted to.

← Previous
Batch Calls - SDK Core

Next - Developers
Automations →

```
import {
  Button,
  Form,
  FormGroup,
  FormControl,
  Spinner,
  Card
} from "react-bootstrap";
import "./batchCall.css";
import { ethers } from "ethers";

let account;
```



Approve the DAI spend, then enter an upgrade amount and the receiver + flowRate for your stream

Keep in mind that this is a short example, you could run many upgrade operations & callAgreement operations in this single transaction if you wanted to.

← Previous
Batch Calls - SDK Core

Next - Developers
Automations →

Last modified 6mo ago

WAS THIS PAGE HELPFUL?

Automations

Use these tools to augment your Superfluid applications

Auto-Wrap
Automatically wrap when Super Token balances get low

Stream Scheduler
Schedule the automatic starting/stopping of streams

Vesting Scheduler
Schedule the automatic vesting of tokens in streams

Super Token balances get low

starting/stopping of streams

vesting of tokens in streams

Stream Accounting API
Get data on streams that's digestible for your accounting

Each page comes with a developer guide on getting started with the service.

← Previous
Batch Calls - Frontend Examples

Next
Auto-Wrap →

Last modified 4mo ago

Auto-Wrap

What's Auto-Wrap?

Auto-Wrap is an automated token wrapping system that automatically wraps ERC20 tokens to Super Tokens just in time to keep your streams running.

When your Super Token balance reaches a certain lower threshold, Auto-Wrap steps in and wraps enough tokens into the needed Super Token on your behalf to ensure you never run out of balance, as that would make all streams stop.

Why Auto-Wrap?

Most organizations don't hold their assets as Super Tokens. DAOs are more likely to hold USDC than USDCx in their treasury to limit exposure to third party protocols. So when a DAO is paying contributors in streams of USDCx they need to ensure they periodically wrap additional USDC in USDCx to keep their streams running.

This can be a tedious manual process because it requires you to:

1. Periodically check the Super Token balance to make sure it isn't reaching zero

2. If is approaching zero, manually wrap additional USDC to USDCx to avoid running out of USDCx balance.

With Auto-Wrap, this manual process is fully automated.

The system keeps monitoring a wallet balance of a specific token (i.e. USDC) and when it's too low (normally less than 2 days worth of outgoing streaming) it automatically wraps a variable amount (normally 7 days worth of outgoing streaming) to keep streams running over time.

All payroll admin needs to do is ensure they're holding enough USDC in their wallet - the wrapping part is taken care of by Auto-Wrap.

Example:

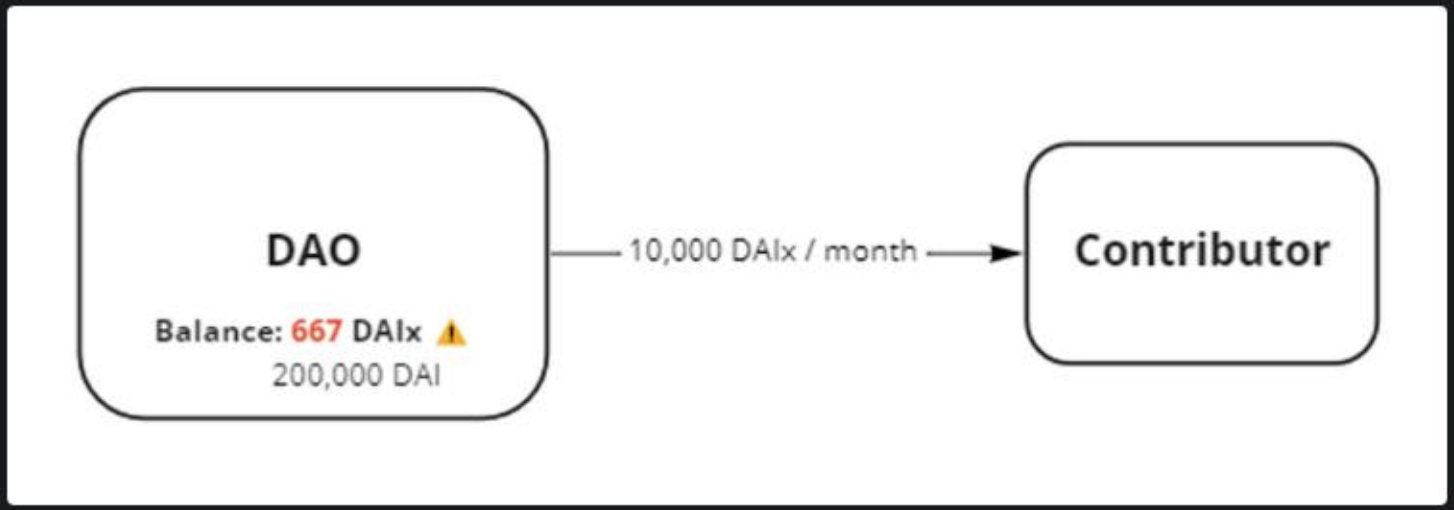
1. Auto-Wrap identifies that the DAO's DAIx balance is running low. In this example, it's less than a two days away from running out of DAIx.

2. Auto-Wrap automatically steps in and replenishes the DAIx balance back to seven days worth of outgoing stream so payment continues without interruption. No manual triggering required!

Setting Up Auto-Wrap

Example:

1. Auto-Wrap identifies that the DAO's DAIx balance is running low. In this example, it's less than a two days away from running out of DAIx.



2. Auto-Wrap automatically steps in and replenishes the DAIx balance back to seven days worth of outgoing stream so payment continues without interruption. No manual triggering required!

Setting Up Auto-Wrap

https://www.notion.so/superfluidhq/Setting-Up-Superfluid-Auto-Wrap-9a565d53bbee4bdc9...

Stream Scheduler

Superfluid's contract template for simplifying automation of stream start and end dates

What's the Stream Scheduler?

The Stream Scheduler allows you to:

1. Schedule when an existing stream is to be closed
2. Schedule when to start a new stream
3. Schedule when to start a new stream and when that new stream is to be closed

This feature can be be used directly in the Superfluid Dashboard after getting approved for access or by making use of the underlying contracts in conjunction with off-chain automations (detailed in documentation here).

Why Schedule Streams?

If you have an intended end date and/or start date for a stream, instead of having to *manually*

trigger the action, you could use the Stream Scheduler to *automatically* trigger the action at the desired time. This is especially useful in the case of streaming payroll, subscriptions and token vesting.

Superfluid streams are perpetual by default - they run until you cancel them or you run out of balance. If you want to be streaming someone tokens for a fixed duration and not have to deal with manually triggering the creation or deletion of the stream, the Stream Scheduler is the solution.

Example:

(view these Miro diagrams here)

1. It's January 1st. You give the Stream Scheduler contract operator permissions and then schedule a stream with the below details:

Flow Rate: 100 DAIx/mo.
To: Alice's account
Start: March 1st, 2023 at 12:00am
End: March 20th, 2023 at 12:00am

2. March 1st comes and the stream is initiated from you to Alice.

3. March 20th comes and the stream from you to Alice is cancelled.

End: March 20th, 2023 at 12:00am

2. March 1st comes and the stream is initiated from you to Alice.

3. March 20th comes and the stream from you to Alice is cancelled.

Setting Up Stream Scheduling

Setting Up Stream Scheduling
Superfluid on Notion

← Previous
Auto-Wrap

Next
Vesting Scheduler →

Vesting Scheduler

What's the Vesting Scheduler?

The Vesting Scheduler smart contract allows you to set up a vesting schedule of tokens for a specific recipient. The Vesting Scheduler contract is non-custodial, it does not hold any tokens, rather it uses permissions to move them from your wallet (or Safe).

The Vesting Scheduler comes with the option to add a cliff date (and amount), after which it vests tokens linearly sending them to the recipient via a by-the-second Superfluid stream (the receiver does not need to claim as these tokens will appear directly in their wallet).

Setting up vesting schedules via the Vesting Scheduled contract will soon be possible directly on the Superfluid Dashboard, however you can already do so now by interacting with the contract directly in conjunction with off-chain automations (detailed in documentation here).

Why Automate Vesting with Superfluid?

Superfluid Protocol allows you to set up a linear vesting schedule with (or without) a cliff while keeping all the to-be-vested tokens in your wallet or Safe (to limit any exposure to our protocol smart contracts).

In a nutshell:

- Set up linear vesting with (or without a cliff)
- Specify both a custom cliff date/time and a custom cliff amount
- Specify your custom total vesting amount and its time framework
- Keep all tokens in your wallet (or Safe) until they are sent directly to the recipient
- I.e. "I'd like to set up a 4 years vesting schedule, that sends 4000 AMZNx linearly with a cliff of 1 year for 1000 AMZNx"

Example:

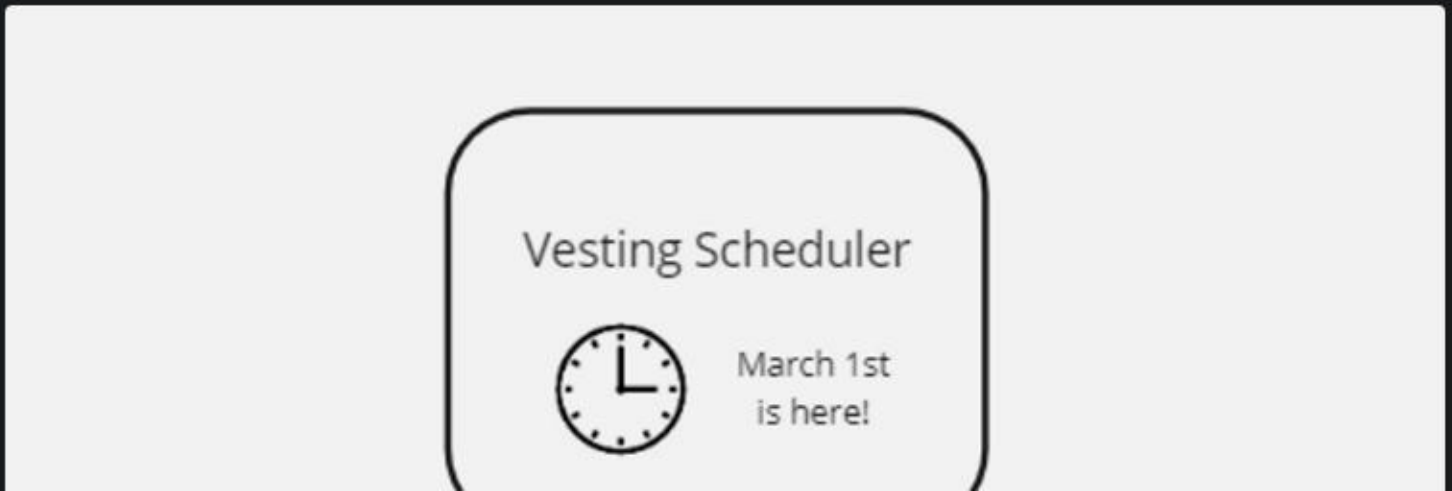
1. It's January 1st. You are essentially looking to vest **400 AMZNx** to Alice **from February 1st to June 1st**, with linear vesting and a one month cliff.

To do so, you schedule vesting for an employee with the below details:

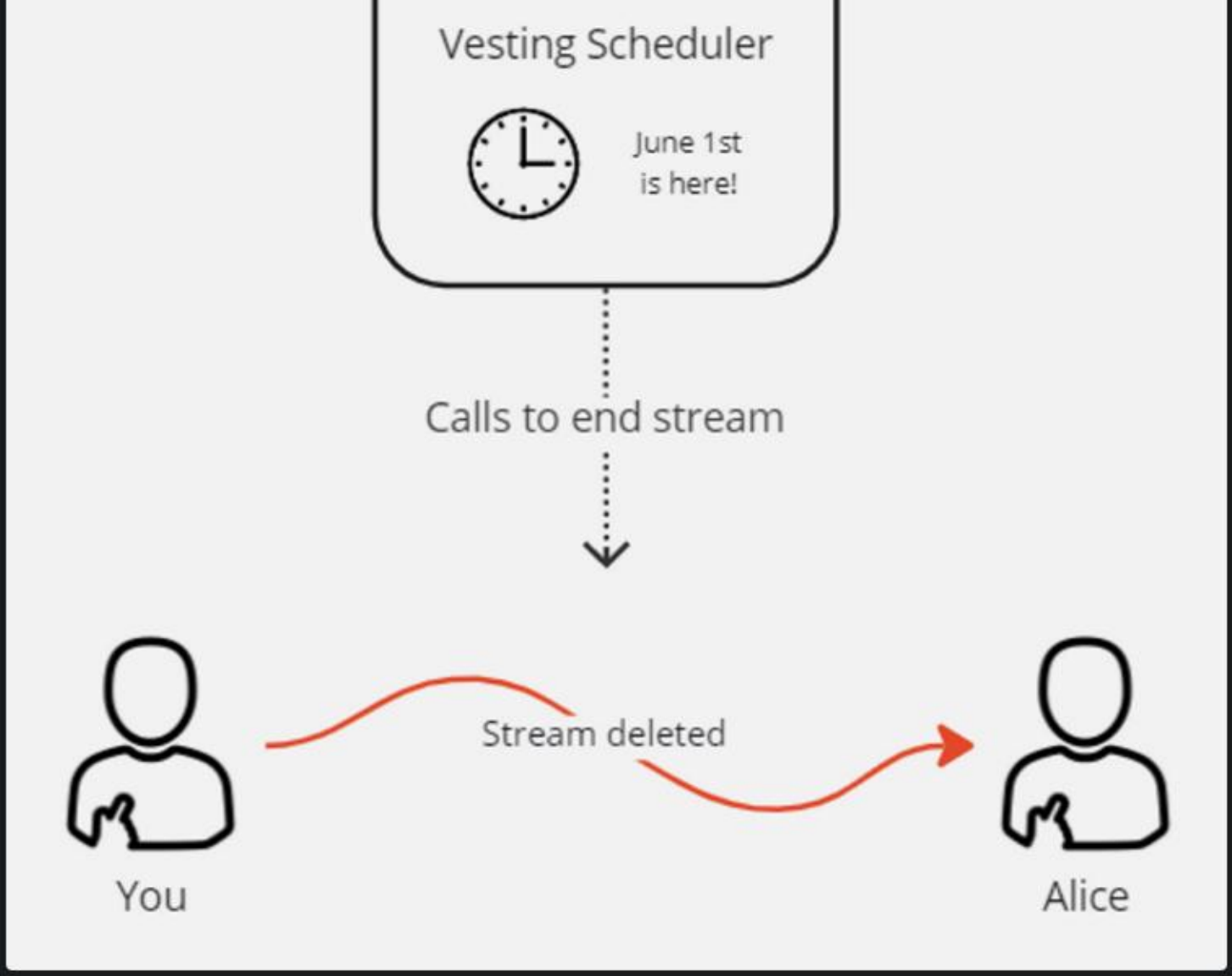
To: Alice's account
Start Date: February 1st
Cliff Date: March 1st
End Date: June 1st
Cliff Amount: 100 AMZNx
Flow Rate: 100 AMZNx/mo.

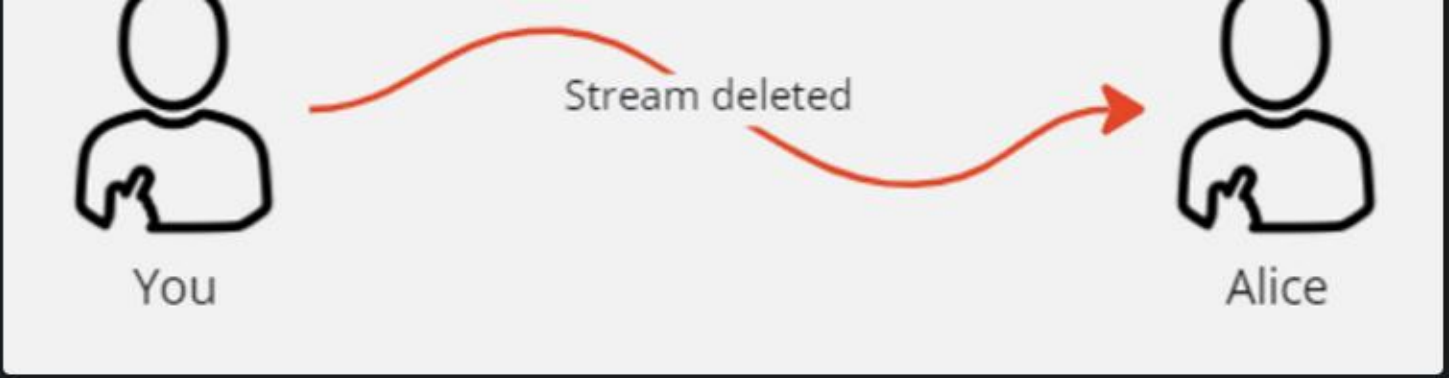
2. It's March 1st - the cliff amount is transferred to Alice and the vesting stream begins

Note: there were no actions to be taken on the Feb. 1st start date as a cliff date was provided



3. It's June 1st and the vesting is complete - the stream is cancelled.


Vesting Scheduler
June 1st
is here!
Calls to end stream
Stream deleted
You
Alice



Setting Up Vesting Scheduling

https://www.notion.so/superfluidhq/Setting-Up-Automated-Vesting-f3e11a257a2d4b0b892...
www.notion.so

← Previous
Stream Scheduler

Next
Stream Accounting API →

Last modified 7mo ago

WAS THIS PAGE HELPFUL?

Stream Accounting API

Overview

Streams move value every second, but accounting tools don't record value transfer on a real-time basis (typically it's monthly). The Stream Accounting API can be used to represent your streams in a manner that's consumable by your traditional accounting tools.

Functionality

The Stream Accounting API is RESTful and exposes a single endpoint which allows for fetching stream data across all Superfluid tokens and networks.

The API lets you **chop up the amounts an address has been streaming over time into accounting periods of your choice** (monthly, daily, even hourly).

Token Pricing

If the token you are streaming has price tracking from Coingecko, then you will be able to get price information for each period in a currency of your choice as well. You can also select price granularity. This lets you choose whether you want to get your prices as an average over a lagging

timeframe of your choice or just the instantaneous price for each period.

Accommodating Flow Rate Updates

The Stream Accounting API accommodates changes in stream flow rates. Basically, a "stream period" is a period over which a stream's flow rate is the same. Once the flow rate is updated, a new stream period begins. When getting accounting data over a certain duration in which the account in concern has a stream whose flow rate has been updated, you'll see that the data is segmented into separate stream periods for each different flow rate.

Outgoing & Incoming

The API accounts for both outgoing and incoming streams in the same data export, denoting incoming streams with positive values and outbound streams with negative values.

All Networks and All Tokens

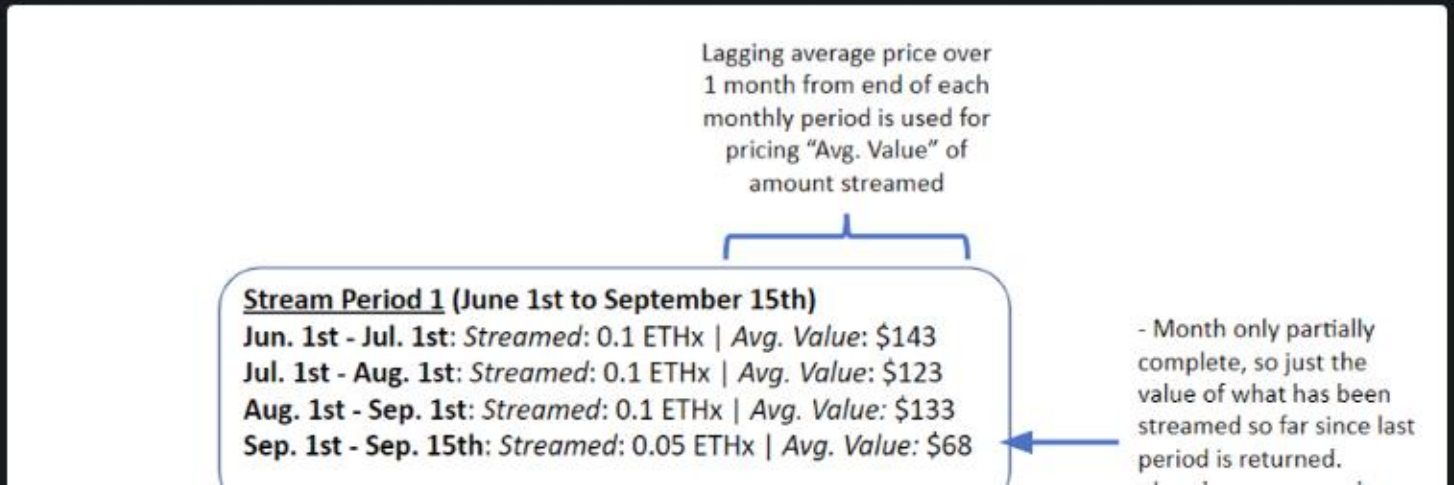
The Stream Accounting API supports all Superfluid-supported networks and tokens through use of the Superfluid Subgraphs.

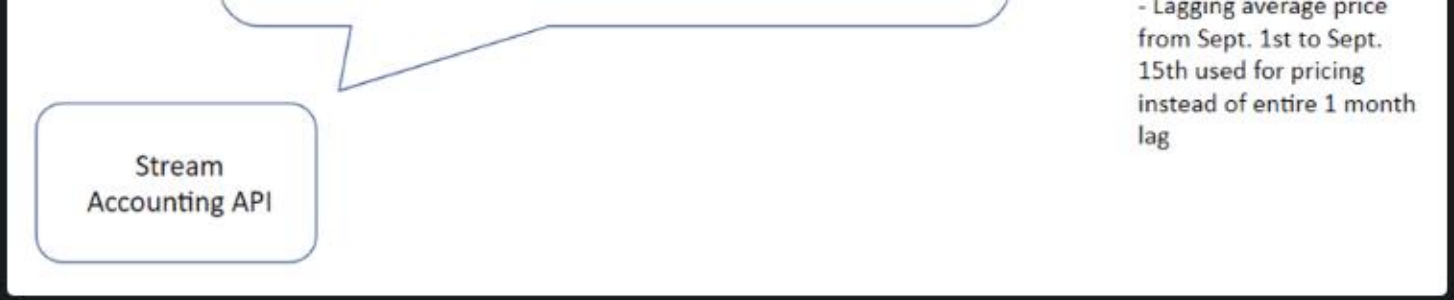
Example

Alice is paying Bob in a stream of 0.1 ETHx per month.

You want to get accounting info on the stream Alice is sending on a monthly basis between **June 1st and September 15th**.

The Stream Accounting API gives you information for the months between June 1st and September 15th.

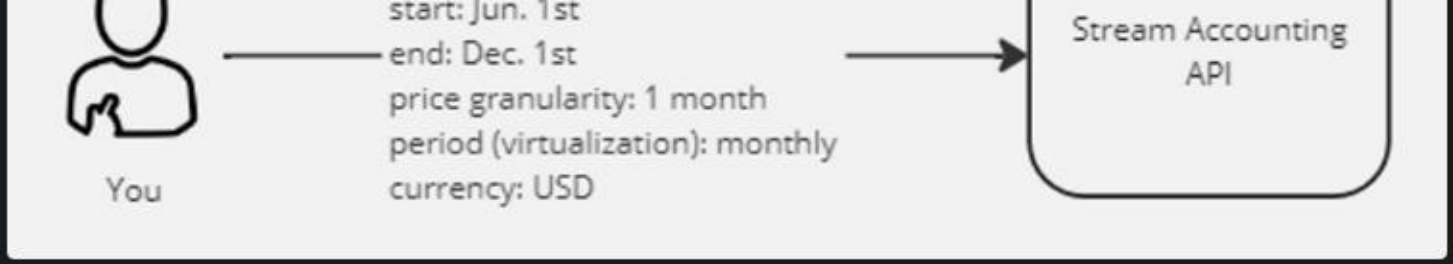




Let's say Alice had updated her stream to 0.2 ETHx/month on September 15th.

Now, you want to get accounting info on the stream Alice is sending on a month basis between June 1st and **December 1st**.

The Stream Accounting API gives you information for the month between June 1st and December 1st.



The Stream Accounting API gives you information for the month between June 1st and December 1st.

Using the Stream Accounting API

Using the Stream Accounting API
Superfluid on Notion

Previous
Vesting Scheduler

Next - Developers
Integration Guides

Deploying a Wrapper Super Token

Let's make a Super Token Wrapper for your ERC20 token

> You can learn more about what Wrapper Super Tokens are HERE.

Anyone can deploy a Wrapper Super Token for any ERC20 token. The deployer account doesn't receive any control or admin powers, and all Super Token logic upgrades are handled by Superfluid Protocol Governance.

Deploying Using Community Deployer Interface

This web app by community member serox.eth provides an easy way to deploy a wrapper

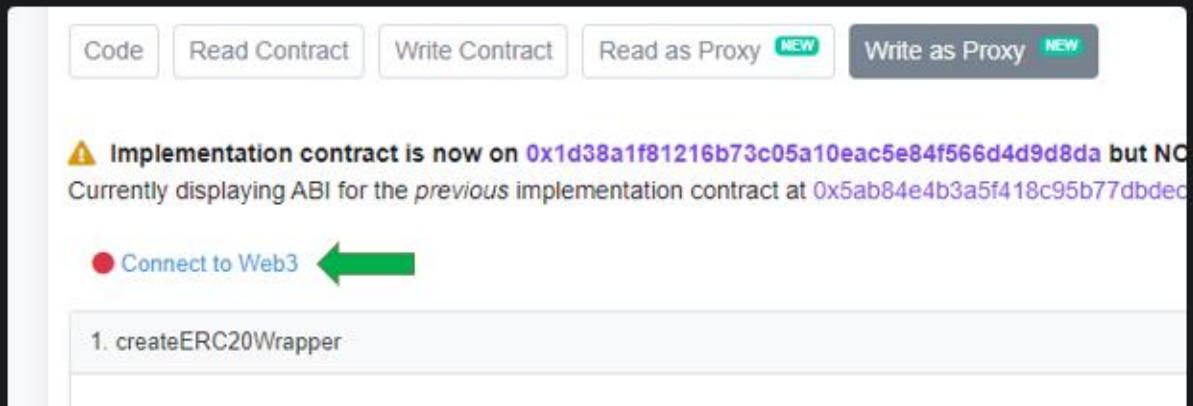
Deploy Supertoken

Deploying Using Block Explorer

1. Head to Super Token Factory contract for appropriate network 👇

- **Polygon**
- **Ethereum Mainnet**
- **Gnosis Chain**
- **Optimism**
- **Arbitrum**
- **BNB Chain**
- **Avalanche**
- **Goerli (testnet)**
- **Mumbai (testnet)**

2. Connect wallet.



underlyingToken (address)

3. Find the `createERC20Wrapper` function. Refer to the below image and provide the required parameters with **no additional characters**.

1. createERC20Wrapper

underlyingToken (address)

Address of token you're making wrapper for

upgradability (uint8)

Put "1" here (no quotes), will give token Superfluid Governance semi-upgradeability

name (string)

"Super" + the current name of your token (ex: Sunny Token → **Super Sunny Token**)

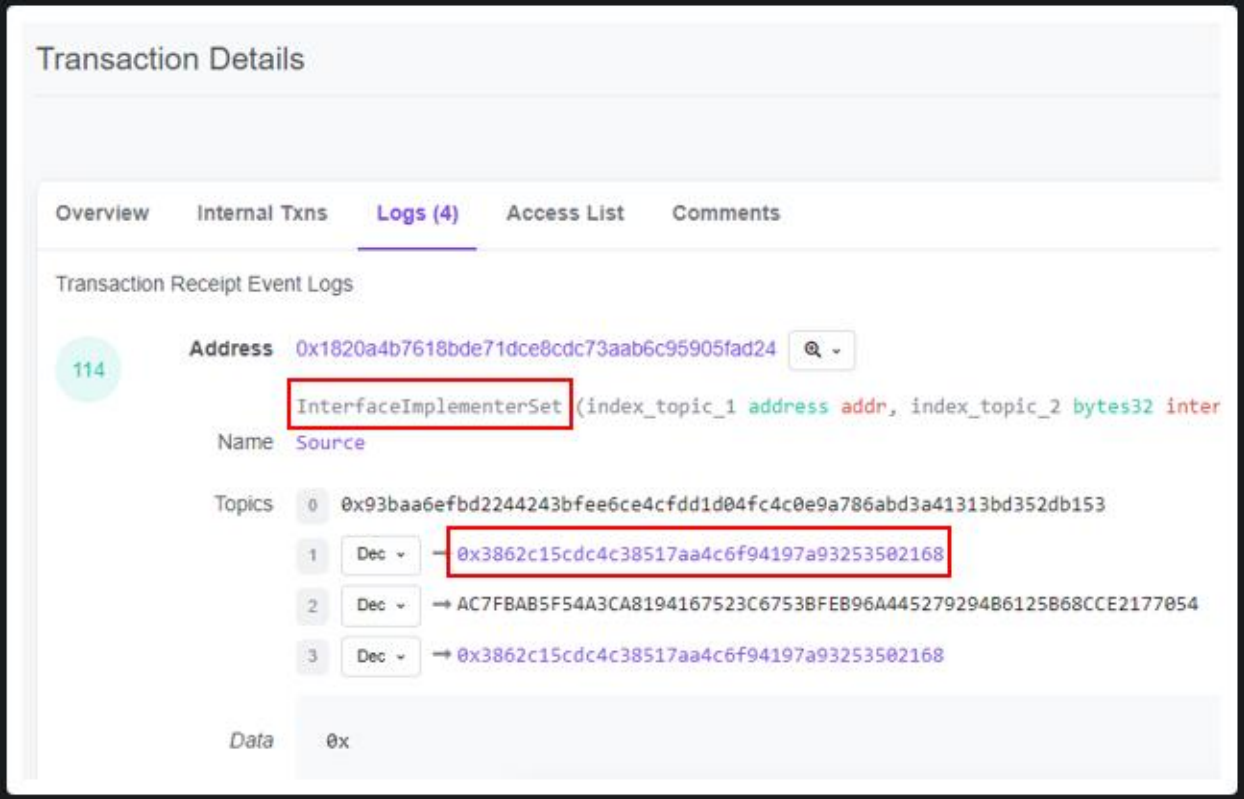
symbol (string)

The current symbol in upper case + lowercase "x" (ex: SUNNY → **SUNNYx**)

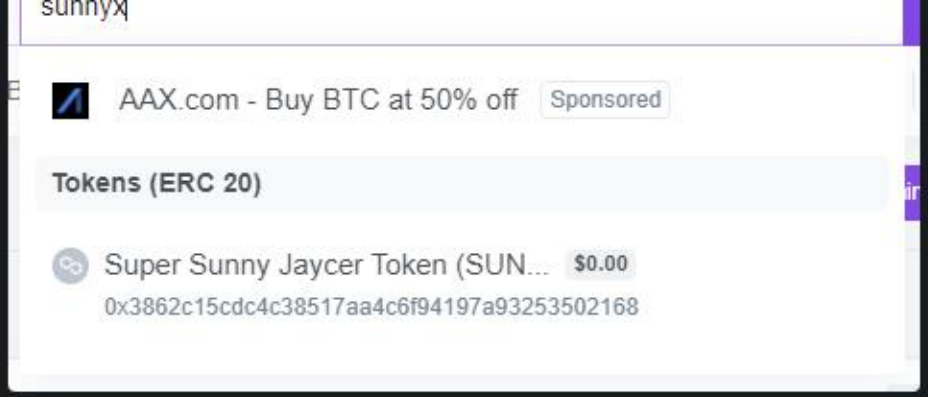
Write

4. Click `Write` and confirm through your wallet. Congrats, you've created a new Wrapper Super Token!

5. To find the address of your new Wrapper Super Token, go to the transaction hash and click on the "Logs" tab. The address is in the "InterfaceImplementerSet" event log (SUNNYx example).



Alternatively, search your token in the top right search bar and find it in the drop down



Deploying Using Deployment Scripts

We've created some handy scripts for deploying the ERC20 Wrapper contract.

```
git clone https://github.com/superfluid-finance/protocol-monorepo/
yarn install --frozen-lockfile
yarn build
cd packages/ethereum-contracts
cp .env.template .env

# edit .env file and configure the correct mnemonic and rpc endpoint
# check truffle-config.js for what environment variables are required
```

Now you can use the `deploy-unlisted-super-token.js` script to deploy the wrapper:

```
RELEASE_VERSION=v1 npx truffle --network <xdai or matic> \
exec ops-scripts/deploy-unlisted-super-token.js : \
<Underlying Token Address> \
<SuperToken Name> \
<SuperToken Symbol>
```

If you are wrapping an existing token, like DAI or USDC, please use suffix "x" in the SuperToken symbol.

Adding Token to Superfluid Dashboard

Congrats on deploying your Super Token! If you'd like to see it listed on the Superfluid Dashboard, check out this page 👇

Token Dashboard Submission

Previous
Deploy a Super Token

Next
Deploying a Pure Super Token

Integration Guides

Helpful Tips for Integrating Superfluid Into an Existing Project

- For Your Integration: The Basics
- Displaying Token Balances
- Useful Queries for Your Integration
- Constructing Links to The Superfluid Dashboard

Previous

Next

For Your Integration: The Basics

Displaying Token Balances

Useful Queries for Your Integration

Constructing Links to The Superfluid Dashboard

Previous
Stream Accounting API

Next
For Your Integration: The Basics

Last modified 7mo ago

For Your Integration: The Basics

A quick overview of primitives you should understand first before building your integration. Most of this stuff you can find in greater detail elsewhere in our docs.

Supporting Super Tokens

Supporting Money Streams

Supporting Instant Distributions

Developers - Previous
Integration Guides

Next
Supporting Super Tokens

A quick overview of primitives you should understand first before building your integration. Most of this stuff you can find in greater detail elsewhere in our docs.

Supporting Super Tokens

Supporting Money Streams

Supporting Instant Distributions

← Developers - Previous
Integration Guides

Next
Supporting Super Tokens →

Last modified 8mo ago

Supporting Super Tokens

Supporting Super Tokens in Your App

> You can learn more about Super Tokens here.

The first thing you'll need to do is support Super Tokens in your application. You can think of Super Tokens as an *extension* of the ERC20 standard. You still get all of the typical ERC20 methods (`transfer`, `approve`, `transferFrom`, `symbol`, etc), but Super Tokens implement additional logic when determining balances.

These tokens store all balance information for each holder, but they're also plugged directly into Superfluid at the protocol level. The way that balances are calculated will reflect all of the *agreements* that the token owner has engaged in. The two agreements currently used within Superfluid are the Constant Flow Agreement (CFA) and Instant Distribution Agreement (IDA).

For reference, the CFA is the agreement which mediates *money streaming*, and the IDA enables *scalable one-to-many token distributions*. You can read more about them in our docs.

While you might initially think of a Super Token as a rebasing token, it isn't one! Any account, when not using any agreements (i.e. not sending or receiving any streams or engaged in any instant distribution agreements), will behave exactly like a vanilla ERC20 token.

distribution agreements), will behave exactly like a vanilla ERC20 token.

> ⓘ 💡 You can find the Super Token contract interface here, and complete docs on how they work here.

3 General Types of Super Tokens

- **ERC20 wrapper tokens:** These are commonly denominated by the 'x' which is appended to the end of the token symbol, such as USDCx and DAIx. These are a wrapper around an existing ERC20 asset.
  - Calling `upgrade()` on this contract will wrap the underlying ERC20 token as a super token: the underlying asset is deposited into the contract and super tokens are burned in return.
  - Calling `downgrade()` on this contract will unwrap the underlying token: the super tokens are burned and the underlying asset is sent back.
- **Native asset wrapper tokens:** These tokens are wrappers around native assets such as AVAX or MATIC. They're also denominated like ERC20 wrappers with the 'x' appended to the end of the token symbol: i.e. MATICx, ETHx. These have the same upgrade/downgrade methods as the ERC20 wrapper contracts, they're just implemented slightly differently
  - Instead of `upgrade()`, you'll use `upgradeByETH()`, and instead of `downgrade()` you'll use `downgradeToETH()`.
  - `upgradeByETH()` takes no parameters - the amount you'd like to upgrade needs to be included as value along w the call (i.e. via `msg.value`).

- `downgradeToETH()` takes one parameter - which is the amount of tokens you'd like to downgrade
- Note that these functions are also calling `upgradeByETH` and `downgradeByETH` on non-ETH native assets like AVAXx and MATICx

- **Pure super tokens:** These are deployed natively as Super Tokens - i.e. without an underlying address at all. Wrapping & unwrapping is not necessary with these tokens because they come out of the box with Super Token functionality.

Super Tokens & ERC777 Methods

Super Tokens also implement many functionalities of ERC777, including functions like `send()` (i.e. `transfer` with a `userData` field, useful for payments as you can add an order id), patterns like using an `operator` (an alternative to approve), and hooks. However, Superfluid distances itself significantly from the canonical ERC777 standard because `transfer()` and `transferFrom()` do not implement hooks. There have been instances where the transfer and transferFrom hooks led to unintended behavior in other protocols, but this isn't something you'll need to concern yourself with when it comes to Superfluid. You still get all the beneficial features of ERC777, without the scary stuff :-)

Getting Super Token Balances

Just like with traditional ERC20 tokens, you can get an account's balance by calling `balanceOf()`. The Super Token implementation of `balanceOf()` will make use of a function named

`realtimeBalanceOf()` under the hood. You can learn more about the `realtimeBalanceOf()` function in our docs, but all you really need to know is that `balanceOf()` will work exactly as expected and take into consideration all of the agreements that a given account is engaged in.

You can get account balances in one of two ways:

1) Calling `balanceOf()` on the contract's ISuperToken interface as you normally would with any other token:

```
function balanceOf(
    address account
)
    public
    view
    override
    returns(uint256 balance)
{
    // solhint-disable-next-line not-rely-on-time
    (int256 availableBalance, , ,) = super.realtimeBalanceOfNow(account);
    return availableBalance < 0 ? 0 : uint256(availableBalance);
}
```

2) Using the Superfluid SDK Core:

```
await usdcx.balanceOf({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

Other Important Things to Know About Super Tokens:

- They'll always have 18 decimals

> 💡 We're working on putting together a canonical super token list which will follow a similar standard as Uniswap's token list & make the process of rendering super token data much easier. In the meantime, you can use our subgraph to query tokens for each network.

← Previous
For Your Integration: The Basics

Next
Supporting Money Streams →

```
await usdcx.balanceOf({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

Other Important Things to Know About Super Tokens:

- They'll always have 18 decimals

> 💡 We're working on putting together a canonical super token list which will follow a similar standard as Uniswap's token list & make the process of rendering super token data much easier. In the meantime, you can use our subgraph to query tokens for each network.

← Previous
For Your Integration: The Basics

Next
Supporting Money Streams →

Supporting Money Streams

Supporting Streams in Your Application

> You can learn more about streaming here.

When an account opens a money stream using the constant flow agreement, they need to define the receiver, the super token being streamed, and the flow rate of that stream. For example, in our sdk, it looks like this:

```
sf.cfaV1.createFlow({
        receiver: “0x123…”,
        superToken: daix.address,
        flowRate: “385802469135802” //equivalent to 1000 dai per month
});
```

> 💡 Note that the flow rate is the number of tokens that the user will send per second to the receiver (denominated in wei: i.e. with 18 decimals)

When this stream is opened, 4 hours worth of that stream is held in escrow by the protocol. This is known as the stream's *buffer*. The buffer is a mechanism to help keep Superfluid secure. With Superfluid streams, you don't need to lock up the entire stream amount up front. You can learn more about why this works the way that it does here, but it's worth keeping that buffer amount in mind.

For example, as long as I have at least 4 hours worth of the stream in my balance, I can open the above stream at 1000/month. Having all 1k tokens in my wallet is unnecessary: I can keep them working in Aave or Compound, and then periodically top up my wallet balance.

An account can be sending and receiving an arbitrary number of streams at any given point in time. However, the thing that will likely matter for your integration is going to be an account's **net flow rate**.

The `netFlowRate` is the total number of tokens being sent or received each second. Accounts will have different net flow rates for each token depending on their streaming activity. As we'll see in the next section, this will be relevant for displaying a balance that is changing in real time. You can get the `netFlowRate` in three different ways:

1) Using the SDK Core:

```
await sf.cfaV1.getNetFlow({
  superToken: string,
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

2) Using the Superfluid subgraph, using an entity such as `accountTokenSnapshot`

```
#this query will return the total netFlowRate for account "0xDCB..." and DAIx on Goerli
query MyQuery {
  accountTokenSnapshots(
    where: {account: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721", token: "0xf2d6889855
  ) {
    token {
      name
      symbol
    }
    totalNetFlowRate
  }
}
```

> Note that you can experiment with your own queries in the subgraph playground in the Superfluid console.

3) You can also get `netFlowRate` data in solidity by calling `getNetFlow` on the CFA contract directly:

```
    token {
      name
      symbol
    }
    totalNetFlowRate
  }
}
```

> Note that you can experiment with your own queries in the subgraph playground in the Superfluid console.

3) You can also get `netFlowRate` data in solidity by calling `getNetFlow` on the CFA contract directly:

Previous
Supporting Super Tokens

Next
Supporting Instant Distributions

Supporting Instant Distributions

Supporting IDA Operations in Your App

> You can learn more about the IDA here.

The instant distribution agreement (IDA) allows users to send one-to-many transfers in a scalable way with a single transaction. The IDA allows a `publisher` to create an `index`, and grant `units` of this index to `subscribers`. Once units are given, a publisher can `distribute` a number of super tokens to the index. These super tokens will be automatically distributed to each subscriber in amounts that are proportional to the number of units each subscriber has been granted relative to the total number of outstanding units.

After funds are distributed, subscribers can call `claim()` to receive all tokens distributed to them so far through that index. Calling `approveSubscription()` will give a user all funds distributed so far, and will allow them to automatically receive all funds distributed in the future.

← Previous
Supporting Money Streams

Next
Gating with Superfluid Subscriptions... →

Supporting IDA Operations in Your App

> ⓘ You can learn more about the IDA here.

The instant distribution agreement (IDA) allows users to send one-to-many transfers in a scalable way with a single transaction. The IDA allows a `publisher` to create an `index`, and grant `units` of this index to `subscribers`. Once units are given, a publisher can `distribute` a number of super tokens to the index. These super tokens will be automatically distributed to each subscriber in amounts that are proportional to the number of units each subscriber has been granted relative to the total number of outstanding units.

After funds are distributed, subscribers can call `claim()` to receive all tokens distributed to them so far through that index. Calling `approveSubscription()` will give a user all funds distributed so far, and will allow them to automatically receive all funds distributed in the future.

← Previous
Supporting Money Streams

Next
Gating with Superfluid Subscriptions... →

Last modified 8mo ago

Gating with Superfluid Subscriptions & Guild.xyz

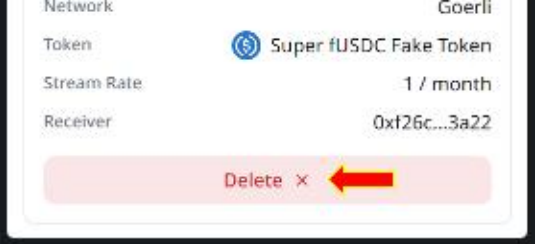
This guide will thoroughly explain how to utilize Superfluid Subscriptions with Guild.xyz. Using the two, you can control access through Guild based on the existence of a Superfluid subscription stream to a desired receiving address.

Quick Links

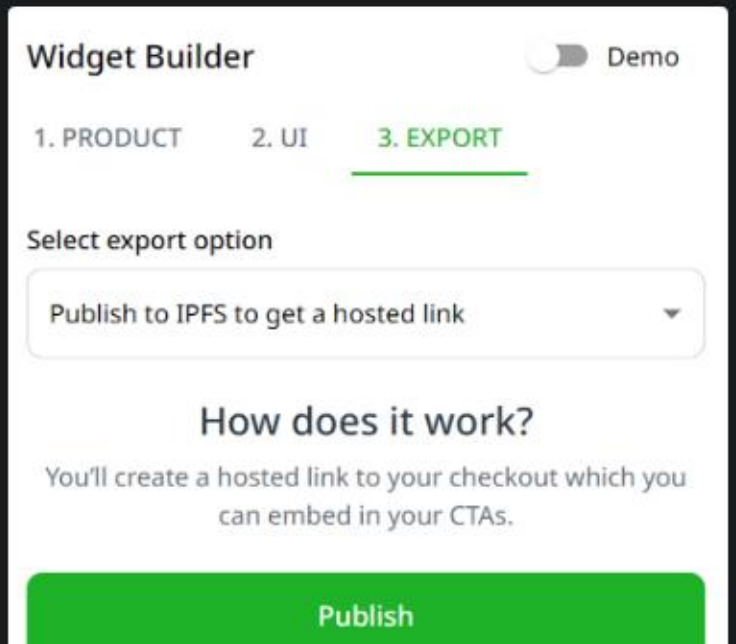
Checkout Builder | Console | Flow Rate Converter | Dashboard | Guild.xyz

1. Create Your Subscription Payment Widget

1. Head to the Checkout Builder
2. Delete the initial payment option set in **Payment Details Summary** at the bottom of the Product section.

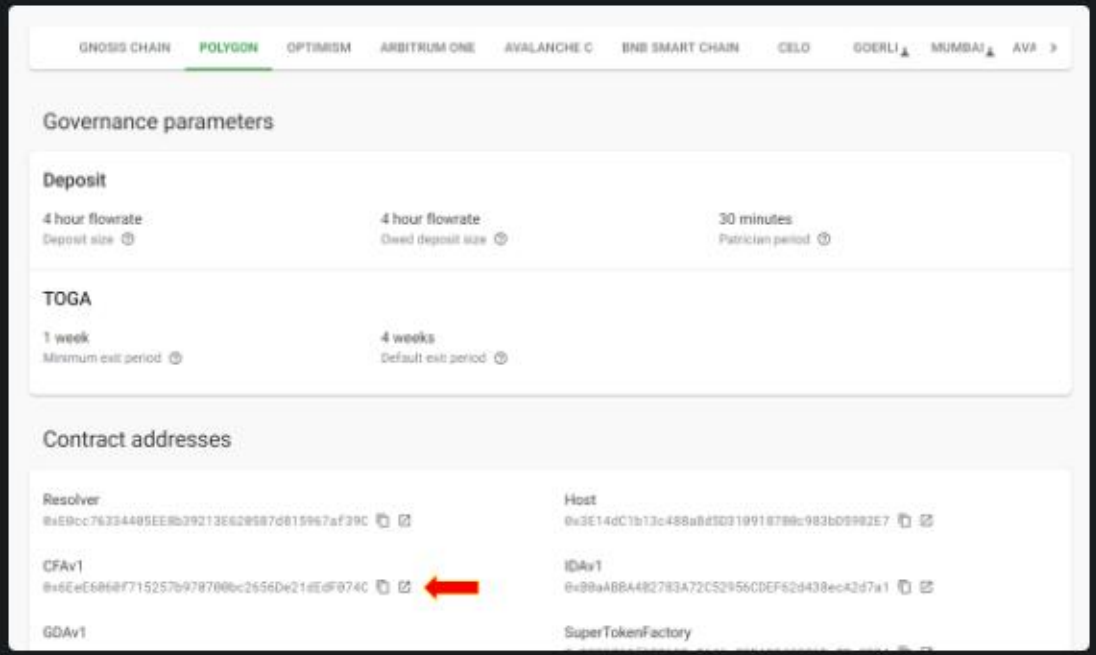


3. Begin customizing your widget payment info on **Product** and aesthetic on **UI**
4. On **Export**, one-click deploy to IPFS and get the payment URL (or save the JSON for later if you want to embed in your site directly)



2. Collect your Addresses

1. Copy down the CFAv1 Address for your intended network of choice from the Console



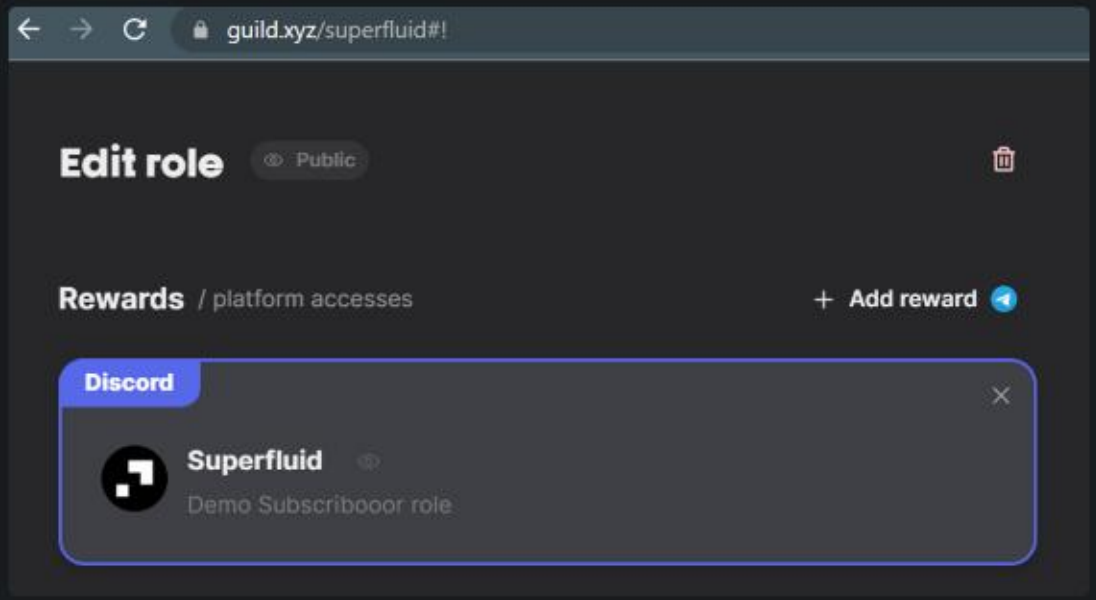
2. Copy down the address of the Super Token you're seeking payment in from the Console. You can look through the list or search the token symbol on the search bar at the top of the page.

3. Get Wei/Sec Flow Rate

Head to our Flow Rate Calculator to convert the monthly rate you elected for your Checkout Widget Payment Option into a wei/sec rate. Copy it down.

4. Create Your Guild Gate

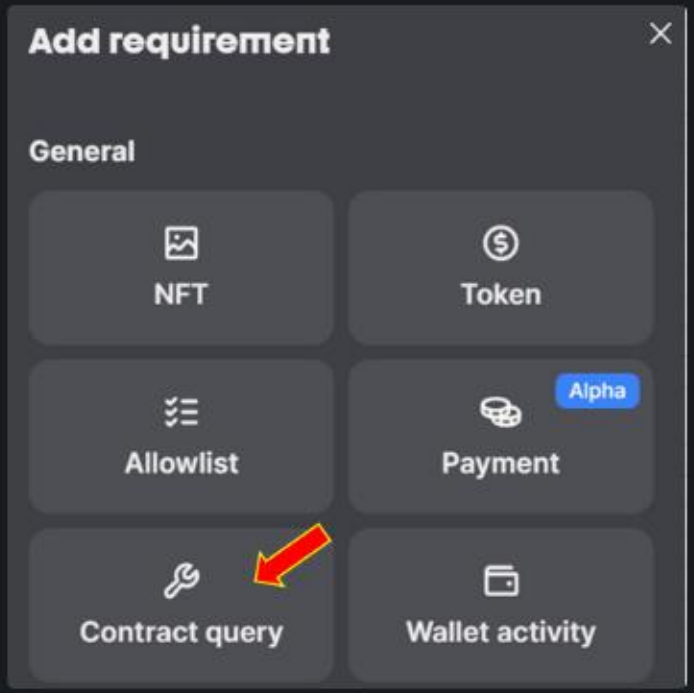
1. Head to Guild.xyz, select what you're looking to gate, and create a role.



Just an example of what it could look like after set up

Just an example of what it could look like after set up

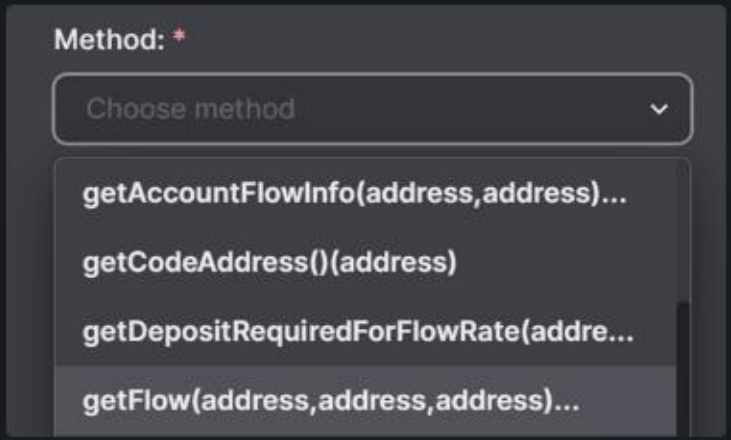
2. Select the requirement as "Contract query"



2. Select the intended network and put in the CFAv1 address copied down from earlier as the Contract Address

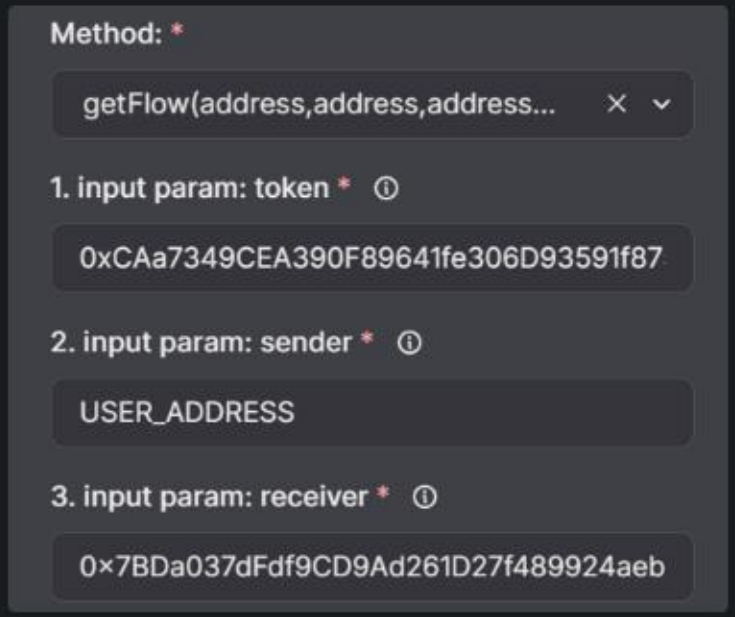
← Add Contract query...

3. In the Method dropdown, select the `getFlow` method

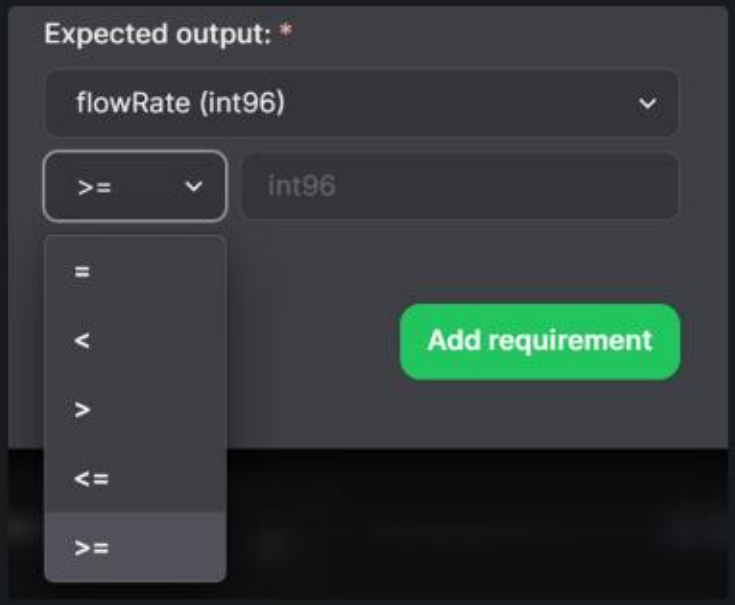


4. Input the following:

- `receiver` - the address to receive the subscription payments (same receiver you set in Payment Options on the Checkout Widget)
- `token` - the address of the Super Token you're accepting payment in (which you copied down earlier)
- `sender` - leave the sender address as USER_ADDRESS



5. Select flowRate in the final drop down and make the qualifier `>=`



6. Input the wei/sec flow rate that you copied down earlier

Start Using! 🏁

Share your Checkout Widget with anyone - people who subscribe with it will be able to access the whatever it is you have gated with Guild.xyz.

Additional Considerations:

Displaying Token Balances

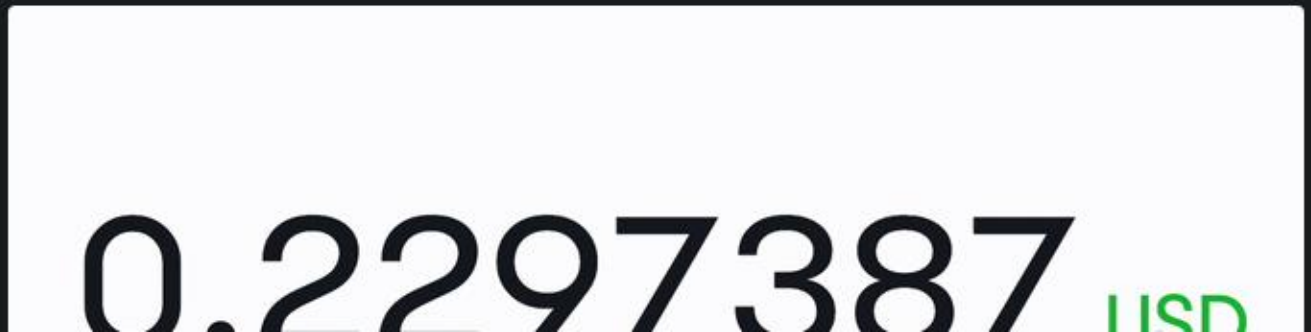
Getting Super Token Balances - Both Static and Dynamic

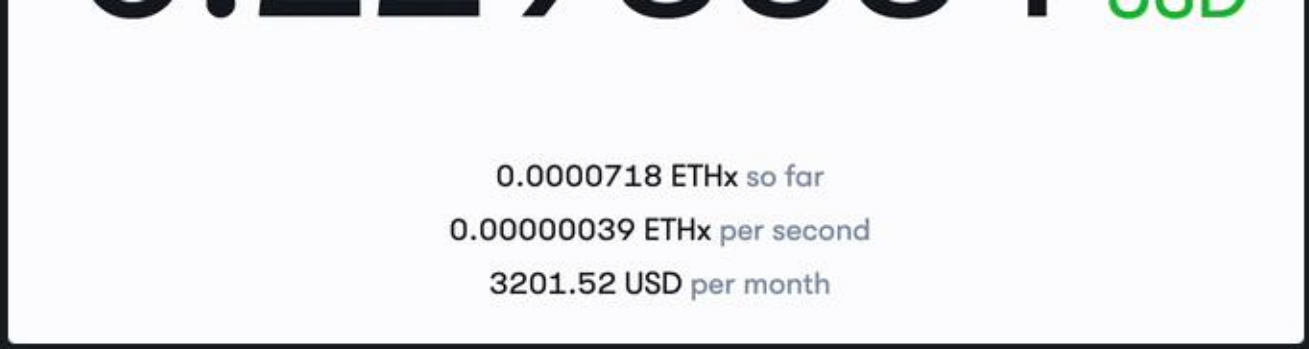
Getting Standard Account Balances

To get an account's balance, you can call `balanceOf()` on each individual Super Token contract, and pass in the address of the account you're looking to query as the lone param. This works just like an ERC20 `balanceOf()` call.

Displaying an Account's Streaming Balance in Your Own App

Our guess is that you want to display something like this in your own app:





we love a good streaming balance

`balanceOf()` will give you the balance of the token holder as of the most recent `block.timestamp` value, but you won't be able to call this function fast enough to render a truly by-the-second real time balance animation in your application. As such, we recommend using a combination of an animation which takes into consideration the static balance of the user, the netflow of the user for each token, and regular polling of the Superfluid subgraph to check for updates to a user's net flow.

What we've seen other teams do is take into consideration the static balance & dynamic balance of each user by listening for `flowUpdatedEvents`, `indexUpdatedEvents`, `transferEvents`, and `upgrade` / `downgrade` events to generate the correct up to date token balance. From here, if a user is actively sending or receiving streams for a token, an animation helps to bring this to life & usually gets really positive reviews.

If you're interested in seeing an example of how the streaming animation works via our console, you should take a look at this component:

combination of an animation which takes into consideration the static balance of the user, the netflow of the user for each token, and regular polling of the Superfluid subgraph to check for updates to a user's net flow.

What we've seen other teams do is take into consideration the static balance & dynamic balance of each user by listening for `flowUpdatedEvents`, `indexUpdatedEvents`, `transferEvents`, and `upgrade` / `downgrade` events to generate the correct up to date token balance. From here, if a user is actively sending or receiving streams for a token, an animation helps to bring this to life & usually gets really positive reviews.

If you're interested in seeing an example of how the streaming animation works via our console, you should take a look at this component:

superfluid-console/FlowingBalance.tsx at master · superfluid-finance/superfluid-cons...
GitHub

Previous
Gating with Superfluid Subscriptions...

Next
Useful Queries for Your Integration

Useful Queries for Your Integration

How to use the Superfluid Subgraph to Get Data Based on Events

Superfluid Events

Whether you're building a wallet, a block explorer, or data platform, you'll need to display up to date account data. This means that you'll need to listen for events where balances change and as stream/distribution operations are performed. You can see a full list of events in our solidity repo, but we recommend taking a close look at the following events as they may prove to be most useful.

← Previous
Displaying Token Balances

Next
Instant Distribution Events →

Last modified 5mo ago

Useful Queries for Your Integration

How to use the Superfluid Subgraph to Get Data Based on Events

Superfluid Events

Whether you're building a wallet, a block explorer, or data platform, you'll need to display up to date account data. This means that you'll need to listen for events where balances change and as stream/distribution operations are performed. You can see a full list of events in our solidity repo, but we recommend taking a close look at the following events as they may prove to be most useful.

← Previous
Displaying Token Balances

Next
Instant Distribution Events →

Last modified 5mo ago

Instant Distribution Events

Query IDA Events

IDA Events (For Distributions)

`indexDistributionClaimed` : emits when a subscriber claims a pending distribution. This will increment the super balance of the subscriber

Solidity Event:

```
    * @param token Super token address
    * @param publisher Index publisher
    * @param indexId The specified indexId
    * @param subscriber The subscriber units updated
    * @param amount The pending amount claimed
    */
  event IndexDistributionClaimed(
      ISuperfluidToken indexed token,
      address indexed publisher,
      uint32 indexed indexId,
      address subscriber,
```

```
uint256 amount);
```

Sample Subgraph Query:

```
#get events where subscriber "0xdcb" claimed a pending distribution
query MyQuery {
  indexDistributionClaimedEvents(
    where: {subscriber: "0xDCB45e4f6762C3D7C61a00e96Fb94ADb7Cf27721"}
  ) {
    amount
    indexId
    timestamp
    token
  }
}
```

`subscriptionApprovedEvents` : emits when a subscribed approves a subscription. If the subscribed has any unclaimed distributions, this will increment the token balance of the subscriber

Solidity Event:

Sample Subgraph Query:

```
#get events where subscriber "0xdcb" approved a subscription
```

```
query MyQuery {
  subscriptionApprovedEvents(
    where: {subscriber: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721"}
  ) {
    timestamp
    token
    transactionHash
  }
}
```

`indexUpdatedEvents`: emits when an index is updated. This will happen when a publisher distributes funds to an index. If a subscriber has approved the subscription by calling `approveSubscription()`, then these events will increment the user's balance. In this case, we'd recommend getting a user's list of index subscriptions and then simply calling `balanceOf()` when an indexUpdatedEvent is emitted for one of those subscriptions.

Solidity Event:

```
/**
 * @dev Index updated event
 * @param token Super token address
 * @param publisher Index updater and publisher
 * @param indexId The specified indexId of the updated index
 * @param oldIndexValue The previous index value
 * @param newIndexValue The updated index value
 * @param totalUnitsPending The total units pending when the indexValue was updated
 * @param totalUnitsApproved The total units approved when the indexValue was updated
```

```
    * @param userData The user provided data
    */
   event IndexUpdated(
       ISuperfluidToken indexed token,
       address indexed publisher,
       uint32 indexed indexId,
       uint128 oldIndexValue,
       uint128 newIndexValue,
       uint128 totalUnitsPending,
       uint128 totalUnitsApproved,
       bytes userData);
```

Sample Subgraph Queries

```
#get a subscriber's index subscriptions
{
   accounts(where: {
   #your address below
    id: "0x..."
    }) {
    subscriptions{
        index {
          token{
            symbol
          }
        }
        id
        approved
```

```
        }
        id
        approved
        units
        totalAmountReceivedUntilUpdatedAt
      }
    }
  }
```

Then, you can listen for events for the above indexIds^

```
query MyQuery {
  indexUpdatedEvents(where: {indexId: $id}) {
    timestamp
  }
}
```


Previous
Useful Queries for Your Integration

Next
Money Streaming Events

```
        }
        id
        approved
        units
        totalAmountReceivedUntilUpdatedAt
      }
    }
  }
```

Then, you can listen for events for the above indexIds^

```
query MyQuery {
  indexUpdatedEvents(where: {indexId: $id}) {
    timestamp
  }
}
```


Previous
Useful Queries for Your Integration

Next
Money Streaming Events

Money Streaming Events

Query CFA Data

CFA Events (For Money Streaming)

`flowUpdated`: emits any time there is a creation, update, or deletion event of a stream. Note that "type" will represent the type of update:

0 = creation event

1 = update event

2 = deletion event

Solidity Event:

```
/**
 * @dev Flow updated event
 * @param token Super token address
 * @param sender Flow sender address
 * @param receiver Flow recipient address
 * @param flowRate Flow rate in amount per second for this flow
 * @param totalSenderFlowRate Total flow rate in amount per second for the sender
```

```
 * @param totalReceiverFlowRate Total flow rate in amount per second for the receiver
 * @param userData The user provided data
 *
 */
event FlowUpdated(
    ISuperfluidToken indexed token,
    address indexed sender,
    address indexed receiver,
    int96 flowRate,
    int256 totalSenderFlowRate,
    int256 totalReceiverFlowRate,
    bytes userData
);
```

Sample Subgraph Query:

```
#returns the new netFlowRate along with all flowUpdated events for a user
query MyQuery {
  flowUpdatedEvents(where: {sender: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721"}) {
    timestamp
    token
                type
    totalSenderFlowRate
                totalAmountStreamedUntilTimestamp
                stream {
              streamedUntilUpdatedAt
            }
  }
```

```
query MyQuery {
  flowUpdatedEvents(where: {sender: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721"}) {
    timestamp
    token
                type
    totalSenderFlowRate
                totalAmountStreamedUntilTimestamp
                stream {
              streamedUntilUpdatedAt
            }
  }
}
```

> **Note** that `totalAmountStreamedUntil` timestamp is useful because it will return the total amount streamed until the state change, and `streamedUntilUpdatedAt` will return the total amount stream since the *previous* state change. Learn more about this here.

← Previous
Instant Distribution Events

Next
Super Token Events →

Deploying a Pure Super Token

Deploying your own Super Token

> You can learn more about what Pure Super Tokens are HERE.

Anyone can deploy a Pure Super Token and add their own custom non-Superfluid logic. This guide will assist you with performing your deployment of a governed Pure Super Token.

1. Set Up the custom-supertokens Repo Locally

Follow the "Setup" in the README to get ready for development 👇

GitHub - superfluid-finance/custom-supertokens
GitHub

2. Select Logic

The main contract directory contains an assortment of examples of Pure Super Tokens contracts with common custom logic.

custom-supertokens/contracts at main · superfluid-finance/custom-supertokens
GitHub

If none of those options fit the exact logic you are hoping for, then don't hesitate to copy down PureSuperToken.sol and begin working in your own logic.

> ⓘ If you do copy down PureSuperToken.sol, don't forget to rename the contract in the .sol file to something unique or you'll have an issue when building the contracts for deployment in the next step.

3. Deploy Pure Super Token

Make a .env file out of the template provided and fill out the appropriate fields for the network you intend to deploy on.

Then follow the "Deployment" steps in the README to deploy and verify your Pure Super Token 👇

GitHub - superfluid-finance/custom-supertokens
GitHub

> ⓘ If you do copy down PureSuperToken.sol, don't forget to rename the contract in the .sol file to something unique or you'll have an issue when building the contracts for deployment in the next step.

3. Deploy Pure Super Token

Make a .env file out of the template provided and fill out the appropriate fields for the network you intend to deploy on.

Then follow the "Deployment" steps in the README to deploy and verify your Pure Super Token 👇

GitHub - superfluid-finance/custom-supertokens
GitHub

← Previous
Deploying a Wrapper Super Token

Next
In-Depth →

Super Token Events

Query Super Token Event Data

Super Token Events

`Transfer` : emitted when super tokens are transferred. Will *decrement* the super token balance of the from address (ie. the same as ERC20 transfer).

Solidity Event:

```
/**
 * @dev Emitted when `value` tokens are moved from one account (`from`) to
 * another (`to`).
 *
 * Note that `value` may be zero.
 */
event Transfer(address indexed from, address indexed to, uint256 value);
```

This comes directly from the Open Zeppelin IERC20 implementation that Super Tokens inherit from

Sample Subgraph Query:

```
#get DAIx transfer events on goerli where "0xdcb..." is the sender
query MyQuery {
  transferEvents(
    where: {from: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721", token: "0xf2d68898557ccl
  ) {
    timestamp
    to {
      id
    }
    value
    transactionHash
  }
}
```

`tokenUpgraded` : emitted when tokens are upgraded (i.e. wrapped). This will *increment* a user's super token balance and *decrement* a user's balance in the underlying token

Solidity Event:

```
/**
 * @dev Token upgrade event
 * @param account Account where tokens are upgraded to
 * @param amount Amount of tokens upgraded (in 18 decimals)
 */
```

```
event TokenUpgraded(
    address indexed account,
    uint256 amount
);
```

Sample Subgraph Query:

```
#query that will return all emitted DAIx upgrade events for "0xdcb…" on goerli
query MyQuery {
  tokenUpgradedEvents(
    where: {account: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721", token: "0xf2d6889855
  ) {
    timestamp
    amount
  }
}
```

`tokenDowngraded` : emitted when tokens are downgraded (i.e. unwrapped). This will *decrement* a user's super token balance, and *increment* their balance of the underlying token

Solidity Event:

```
/**
 * @dev Token downgrade event
 * @param account Account whose tokens are upgraded
 * @param amount Amount of tokens downgraded
```

```
 * @param amount Amount of tokens downgraded
 */
event TokenDowngraded(
    address indexed account,
    uint256 amount
);
```

Sample Subgraph Query:

```
#query that will return all emitted DAIx downgrade events for "0xdcb…" on goerli
query MyQuery {
  tokenDowngradedEvents(
    where: {account: "0xdcb45e4f6762c3d7c61a00e96fb94adb7cf27721", token: "0xf2d6889855
  ) {
    timestamp
    amount
  }
}
```


← Previous
Money Streaming Events

Next
Other Helpful Queries →

Other Helpful Queries

Other useful queries for your integration

Other Helpful Queries

How can I query month-on-month stats for Superfluid stream senders? e.g. monthly amount streamed, number of streams started/stopped, etc?

> 💡 NOTE: you probably want to define what start/end of the month means and be consistent and transparent about it: e.g. 12AM of the first of month 11:59PM on the last day of the month.

`ATS = AccountTokenSnapshot`

- it requires queries at the start and end of a month and a set of sender addresses and tokens they are interested in.
- find the `block.timestamp` for the months you're interested in
- execute a query for `ATS` (s) for a set of senders and token at the start of a month/end of a month (timestamps)

month (timestamps).

- you could then get the delta between the two entities (end date `ATS` - start date `ATS`) on the two queries on the follow fields of `AccountTokenSnapshot`
- `totalNumberOfActiveStreams`
- `totalNumberOfClosedStream`
- `totalAmountStreamedInUntilUpdatedAt`
- for this last property, you need to do a check for both the start and end of month timestamp values you used for the query and check if it is equal to `updatedAtTimestamp` (it's pretty much almost never going to be)
- and you'll need to calculate the actual `totalAmountStreamedIn`
- e.g. if `totalAmountStreamedUntilUpdatedAt = 1`, `updatedAtTimestamp = 10` and `start of month timestamp = 15` and `outflowRate = 1`, your actual `totalAmountStreamedIn` for start of month is `6`
- You'll have to do the same for the end of the month

← Previous
Super Token Events

Next
Constructing Links to The Superfluid... →

Last modified 8mo ago

- you could then get the delta between the two entities (end date `ATS` - start date `ATS`) on the two queries on the follow fields of `AccountTokenSnapshot`
- `totalNumberOfActiveStreams`
- `totalNumberOfClosedStream`
- `totalAmountStreamedInUntilUpdatedAt`
- for this last property, you need to do a check for both the start and end of month timestamp values you used for the query and check if it is equal to `updatedAtTimestamp` (it's pretty much almost never going to be)
- and you'll need to calculate the actual `totalAmountStreamedIn`
- e.g. if `totalAmountStreamedUntilUpdatedAt = 1`, `updatedAtTimestamp = 10` and `start of month timestamp = 15` and `outflowRate = 1`, your actual `totalAmountStreamedIn` for start of month is `6`
- You'll have to do the same for the end of the month

← Previous
Super Token Events

Next
Constructing Links to The Superfluid... →

Last modified 8mo ago

Constructing Links to The Superfluid Dashboard

Generating pre-filled dashboard links for your users

How do I create a prefilled Superfluid Dashboard Send Stream link?

If you want to send a payment request using the Superfluid Dashboard, you can do so by tweaking the URL parameters

If you're interested in sending someone else a prefilled Send Stream link, here's how:

A fully-filled send stream link looks as follows:

```
<https://app.superfluid.finance/send?network=polygon&receiver=0x7BDa037dFdf9CD9Ad261D27
```

You may tweak the parameters as follows:

`network` : The intended network of the stream to be created. Examples: `gnosis` , `polygon` , `polygon-mumbai` , `avalanche-fuji` , `optimism` , `arbitrum-one` , `avalanche` , `bsc` , `goerli`

`receiver` : The hex address of the intended recipient of the stream to be created.

`token` : The hex address of the Super Token to be used for the stream. You can find Super Token addresses in the Superfluid Console.

`flow-rate` : The flow rate of the intended stream, `number/frequency` e.g. `1/second`, `1000/month`, `120000/year`

How to construct links to Superfluid Dashboard Stream Details Pages

Method 1 (recommended):
https://app.superfluid.finance/stream/polygon/0xd964ab7e202bab8fbaa28d5ca2b2269a5497cf68-0x7bda037dfdf9cd9ad261d27f489924aebbce71ac-0x3862c15cdc4c38517aa4c6f94197a93253502168-0.0

`sender-receiver-tokenaddress-StreamRevision.revisionIndex`

Method 2 (legacy):
https://app.superfluid.finance/stream/polygon/0x2214a32b03a13a68f5f3d84c17c9fd549ab6fcc01b96ffcea8cf209982cf32da-95

`txhash-logindex`

How do I create a prefilled Superfluid Dashboard Li.Fi Bridge link?

Instructions for a prefilled link to Swap and Bridge in the Superfluid Dashboard

Instructions for a prefilled link to Swap and Bridge in the Superfluid Dashboard

A fully prefilled Li.Fi Bridge Widget link looks as follows:

```
<https://app.superfluid.finance/bridge?fromChain=137&toChain=10&fromToken=0xCAa7349CEA3
```

You may tweak the parameters as follows:

`fromChain` : The chainId of the source chain. e.g. `137` for Polygon

`toChain` : The chainId of the destination chain. e.g. `10` for Optimism

`fromToken` : The token hex address of the source token

`toToken` : The token hex address of the destination token

`fromAmount` : The amount of tokens to be swapped

← Previous
Other Helpful Queries

Next - Developers
Superfluid Subscriptions →

Instructions for a prefilled link to Swap and Bridge in the Superfluid Dashboard

A fully prefilled Li.Fi Bridge Widget link looks as follows:

```
<https://app.superfluid.finance/bridge?fromChain=137&toChain=10&fromToken=0xCAa7349CEA3
```

You may tweak the parameters as follows:

`fromChain` : The chainId of the source chain. e.g. `137` for Polygon

`toChain` : The chainId of the destination chain. e.g. `10` for Optimism

`fromToken` : The token hex address of the source token

`toToken` : The token hex address of the destination token

`fromAmount` : The amount of tokens to be swapped

← Previous
Other Helpful Queries

Next - Developers
Superfluid Subscriptions →

Superfluid Subscriptions

Learn how to add streaming subscriptions features to your apps with Superfluid

Building Subscriptions with Superfluid

When building a subscriptions product on Superfluid, we expect that you'll need a few things

1. You'll need to enable users to create a subscription stream, update that stream if they change their plan, or delete that stream if they opt to cancel -> our widget offers these features off-the-shelf, check out **Superfluid Checkout Widget**
2. You may need to validate whether an address is currently streaming to you, and perhaps get a list of all addresses currently streaming to you -> see # **Getting Stream Data**
3. You'll need to know when a user deletes a stream or updates their plan so that you may make the proper changes to their account. -> see # **Events**

There are many ways that this process can be managed, and your implementation will depend on the needs of your users. With that said, here are a few concepts & code examples that should help you in your development.

Building Subscriptions with Superfluid

When building a subscriptions product on Superfluid, we expect that you'll need a few things

1. You'll need to enable users to create a subscription stream, update that stream if they change their plan, or delete that stream if they opt to cancel -> our widget offers these features off-the-shelf, check out **Superfluid Checkout Widget**
2. You may need to validate whether an address is currently streaming to you, and perhaps get a list of all addresses currently streaming to you -> see # **Getting Stream Data**
3. You'll need to know when a user deletes a stream or updates their plan so that you may make the proper changes to their account. -> see # **Events**

There are many ways that this process can be managed, and your implementation will depend on the needs of your users. With that said, here are a few concepts & code examples that should help you in your development.

← Previous
Constructing Links to The Superfluid...

Next
Superfluid Checkout Widget →

Last modified 1mo ago

Superfluid Checkout Widget

Using the Superfluid Checkout Widget for Web3-Native Subscriptions

> NOTE: **the checkout widget is in open alpha** and is subject to change. We are iterating quickly and are open to feedback!

What is the Superfluid Checkout Widget?

The Superfluid **Checkout Widget** demonstrates a clean pattern for subscription UX in web3 using Superfluid, including all the core web3 interactions needed. Here are some highlights:

- It is minimally-opinionated and uses the most up to date web3 front end tooling
- It supports all Superfluid tokens and networks by default
- It automatically becomes a 1-click checkout if a payer already holds Super Tokens
- It can be styled as a widget, a full page view, or a modal itself
- Fonts, shapes, colors are all customizable

The widget is meant to be imported into existing Wagmi front ends using a React or Vue

The widget is meant to be imported into existing Wagmi front ends using a React or Vue component. You can import the widget library, include JSON that defines your widget's behavior in the props of the subscription checkout component, and your widget will behave as expected. You can create your own custom JSON with our no code widget builder:

Building Your Own Custom Checkout Widget

> Note: if you're looking to get started quickly, check out the examples below which will work with Web3Modal or RainbowKit.

To implement your checkout widget, you'll need to install the widget library, wagmi, and the Superfluid token list from your favorite package manager:

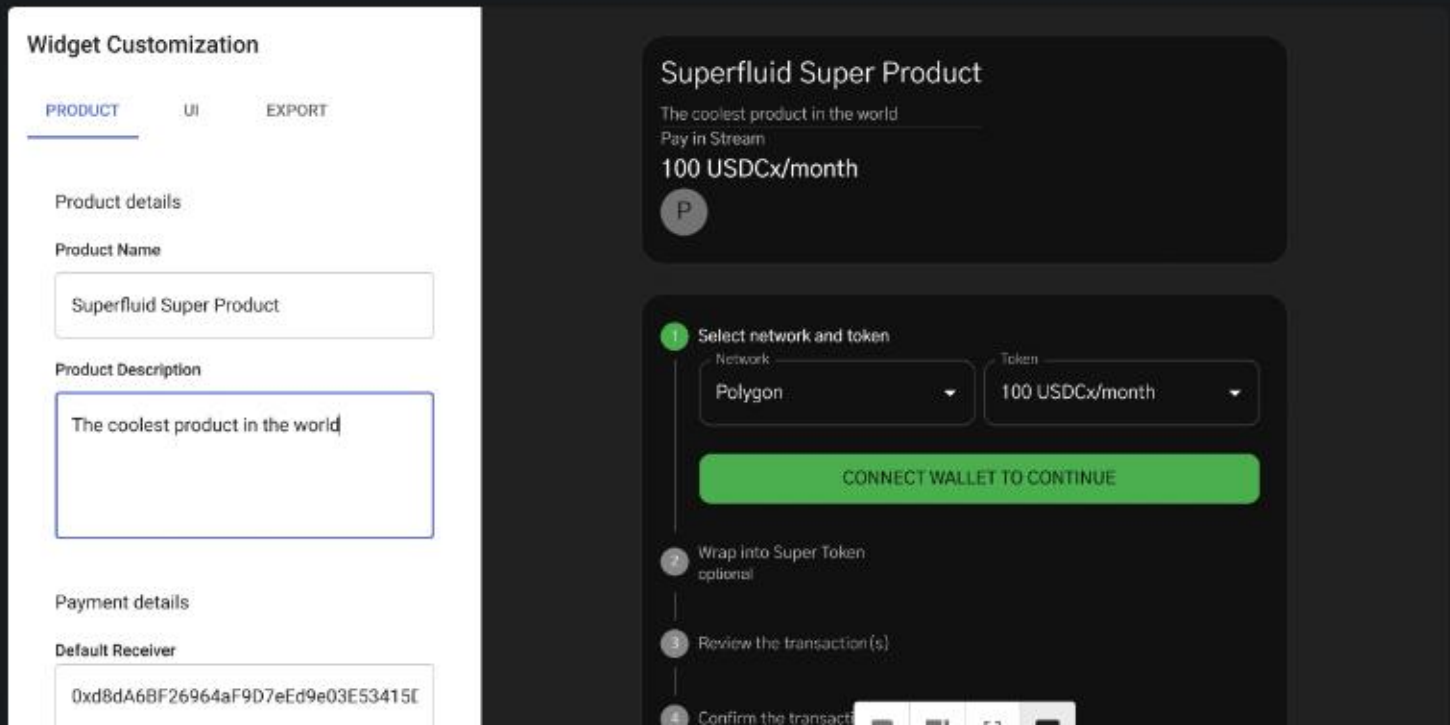
```
#npm
npm install --save @superfluid-finance/widget wagmi @superfluid-finance/tokenlist

#yarn
yarn add @superfluid-finance/widget wagmi @superfluid-finance/tokenlist
```

You can then import the widget into the component you'd like to render it in

```
import { WagmiConfig } from "wagmi";
import SuperfluidWidget from '@superfluid-finance/widget';
```

Before you include the widget in your file, you'll need to generate a Widget JSON that you'll pass as a prop to the widget component. This Widget JSON specifies most of your widget's properties, and you can use our hosted, no code widget builder to visually see what your widget will look like. You can find the widget builder here.

Superfluid Checkout Builder | Web3 Subscriptions Toolkit



The widget builder in action.

User Data in the Widget JSON

Once you have your JSON, you add the file to your project, import it, and use the javascript spread operator to pass it as props to your widget component. See `{...data}` here:

```
import { WagmiConfig } from "wagmi";
import SuperfluidWidget from '@superfluid-finance/widget';
import superTokenList from "@superfluid-finance/tokenlist";


export function MyComponent() {

 return (
   <div>
    <WagmiConfig config={wagmiConfig}>
        <SuperfluidWidget
          {...data}
          tokenList={superTokenList}
          type="dialog"
          walletManager={walletManager}
        >
```

```
        {({ openModal }) => (
          <button onClick={() => openModal()}>Open Superfluid Widget</button>
        )}
        </SuperfluidWidget>
      </WagmiConfig>
    </div>
  )
```

You'll also need a few more props: `tokenList`, `type`, and `walletManager`.

The `tokenList` allows the widget to support all listed Super Tokens. You should have the Superfluid token list already installed it if you used the above npm or yarn commands! You'll just need to include it as a prop like this:

```
tokenList={superTokenList}
```

The `type` prop allows us to specify what kind of widget we want to create. We have a few supported types:

- drawer
- dialog
- full-screen

Finally, the `walletManager` prop is necessary to ensure that the widget works properly with wallet connection libraries such as RainbowKit, Web3Modal, & more. It is recommended that you use the `useMemo` hook from react to set the `walletManager` variable & associated prop:

```
const walletManager = useMemo(
  () => ({
    open,
    isOpen,
  }),
  [open, isOpen]
);
```

Checkout Widget Examples

All examples can be found in this repo.

RainbowKit Example:

widget/examples/widget-vite-react-rainbowkit at master · superfluid-finance/widget
GitHub

Web3Modal Example:

widget/examples/widget-vite-react-web3modal at master · superfluid-finance/widget
GitHub

widget/examples/widget-vite-react-web3modal at master · superfluid-finance/widget
GitHub

IPFS Hosted + Full Page Example:

widget/apps/hosted-widget at master · superfluid-finance/widget
GitHub

Reference Documentation

cfAv1ForwarderAddress | @superfluid-finance/widget

Developers - Previous
Superfluid Subscriptions

Next
Implementing Subscriptions in your ...

Last modified 1mo ago

WAS THIS PAGE HELPFUL?

widget/examples/widget-vite-react-web3modal at master · superfluid-finance/widget
GitHub

IPFS Hosted + Full Page Example:

widget/apps/hosted-widget at master · superfluid-finance/widget
GitHub

Reference Documentation

cfAv1ForwarderAddress | @superfluid-finance/widget

Developers - Previous
Superfluid Subscriptions

Next
Implementing Subscriptions in your ...

Last modified 1mo ago

WAS THIS PAGE HELPFUL?

Implementing Subscriptions in your App

Reference material for building subscription products on Superfluid

Write Operations

Unless you have a very specific reason to use a smart contract or a deeper integration for these write operations, we recommend that you use our **Superfluid Checkout Widget** toolkit. If you prefer to use the SDK. You can find several examples of our SDK in action with front end examples here, and a node module example here.

Getting Stream Data

There are two ways to get stream data: via our SDK and our Subgraph. If you only want to check whether a single address is streaming to you, then our SDK is the simplest way to get this data is to load the token you'd like to use like this.

> ⓘ Note that tokens may be loaded by symbol or by address

```
// Some code
const daix = await sf.loadSuperToken("DAIx");
let flowInfo = await daix.getFlow({
    sender: string,
    receiver: string,
    providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

But if you want to get a list of all addresses streaming to you, then our subgraph is going to be best for this. This is also our recommended method for querying data on a subset of those addresses based on specific criteria (i.e. all addresses with an existing flow rate about x amount).

You can workshop different subgraph queries at console.superfluid.finance/subgraph and use Apollo Client or a similar framework to actually run these queries inside of your application.

Here's an example which will give you all streams that are currently open to your address:

```
//graphql query
query MyQuery {
    streams(where: {receiver: "YOUR_ADDRESS_HERE", currentFlowRate_gt: "0"}) {
```

```
        currentFlowRate
        token {
            symbol
        }
        sender {
            id
        }
        receiver {
            id
        }
    }
}
```

Events

There is more than one way to handle updates to subscriptions. However, it may be useful for you to listen for specific events that are thrown on chain when streams are created, updated, or deleted. The event you're looking for here is the flowUpdated event. You can get flowUpdated events from our subgraph, or query them on chain if you'd like. If you want to get these events from the subgraph, then you can do so with a query like this:

```
// graph query
query MyQuery {
  flowUpdatedEvents(
    where: {receiver: "MY_ADDRESS"}
  ) {
```

```
    sender
    token
    flowRate
    timestamp
  }
}
```

Simple Flow Authentication Example

If you're interesting in seeing this all together, here is an example of an Express app that checks whether or not a stream exists to a specific address before returning 200 (if the stream does not exist, a 401 error is returned).

superfluidInfraPayments/index.js at master · saflamini/superfluidInfraPayments
GitHub

Previous
Superfluid Checkout Widget

Next - Developers
SDK Core General

```
    token
    flowRate
    timestamp
  }
}
```

Simple Flow Authentication Example

If you're interesting in seeing this all together, here is an example of an Express app that checks whether or not a stream exists to a specific address before returning 200 (if the stream does not exist, a 401 error is returned).

superfluidInfraPayments/index.js at master · saflamini/superfluidInfraPayments
GitHub

Previous
Superfluid Checkout Widget

Next - Developers
SDK Core General

Last modified 1mo ago

WAS THIS PAGE HELPFUL?

SDK Core General

Interact with Superfluid using Javascript/Typescript

The Superfluid SDK-Core is a wrapper library around `@superfluid-finance/ethereum-contracts` which allows web developers to interact with the Superfluid contracts. Under the hood, SDK-Core leverages TypeScript, ethers.js and The Graph and GraphQL.

The following pages in this section provide an overview on how to use the SDK core to interact with the protocol and the Interactive Tutorials section will give you live code examples you can use in your own projects.

Previous
Implementing Subscriptions in your ...

Next
Initialization

Last modified 5mo ago

SDK Core General

Interact with Superfluid using Javascript/Typescript

The Superfluid SDK-Core is a wrapper library around `@superfluid-finance/ethereum-contracts` which allows web developers to interact with the Superfluid contracts. Under the hood, SDK-Core leverages TypeScript, ethers.js and The Graph and GraphQL.

The following pages in this section provide an overview on how to use the SDK core to interact with the protocol and the Interactive Tutorials section will give you live code examples you can use in your own projects.

Previous
Implementing Subscriptions in your ...

Next
Initialization

Last modified 5mo ago

Initialization

How to initialize the SDK Core

Required Imports

To install the Superfluid SDK Core, you can run

You'll need to have a project which already includes `graphql` and `ethers` as dependencies.

From here, you can install the Superfluid SDK Core like so:

```
npm install @superfluid-finance/sdk-core
```

Peer Dependencies

You'll also need to install `graphql` and `ethers.js` to work with the SDK.

```
npm install graphql ethers
```

Initializing the SDK Core

When creating the framework, you'll use `Framework.create()` like so:

```
const sf = await Framework.create({
  chainId: Number, //i.e. 137 for matic
  provider: ethersProvider // i.e. the provider being used
});
```

Framework Options

When creating the framework, you must pass in the following options. The only required params are the `provider` and a `chainId`

`chainId: number` - the chain Id for the network

`provider: SupportedProvider` - the provider being used

The Signer Class [Optional]

Web3Provider Signer Example

Below is an example of using the `Web3Provider` object to create a signer. This will likely be the

Below is an example of using the `Web3Provider` object to create a signer. This will likely be the way that most client-side applications create a signer.

```
import { Framework } from "@superfluid-finance/sdk-core";
import Web3Modal from "web3modal";
import { Web3Provider } from "@ethersproject/providers";

// web3Modal example
const web3ModalRawProvider = await web3Modal.connect();
const web3ModalProvider = new Web3Provider(web3ModalRawProvider, "any");

const sf = await Framework.create({
  chainId: 137, //your chainId here
  provider: web3ModalProvider,
});

const web3ModalSigner = sf.createSigner({ web3Provider: web3ModalProvider });

// MetaMask example
const metamaskProvider = new Web3Provider(window.ethereum);
const metaMaskSigner = sf.createSigner({ web3Provider: metamaskProvider });
```

Hardhat Signer Example

Below is an example of creating a signer in a `Hardhat` + `ethers.js` environment. This will likely be the way that the `sdk-core` is used in a testing environment.

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "hardhat";

const sf = await Framework.create({
  chainId: 137, //your chainId here
  provider: ethers.provider,
});

const signer = sf.createSigner({
  privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>",
  provider: ethers.provider,
});
```

Signer/Wallet Example

Below is an example of creating a signer passing in a signer object (this can be a wallet for example). This will likely be the way that the `sdk-core` is used in a Node.js environment (back-end) or a testing environment.

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";

const provider = new ethers.providers.InfuraProvider(
  "matic",
```

```
  "<INFURA_API_KEY>"
);

const wallet = new ethers.Wallet(
  "cf2bea4c6aad8dbc387d5dd68bf408999b0b1ee949e04ff1d96dd60bc3553a49",
  provider
);

const sf = await Framework.create({
  chainId: 137, //your chainId here
  provider,
});

const signer = sf.createSigner({ signer: wallet });
```

Operations

The `Operation` class is an object that is returned after you execute a contract call from this package - instead of immediately executing, we return the `Operation` class which can be either executed to broadcast the transaction or used to create and execute a `BatchCall`. We'll make use of this class in our next sections.

```
import { Framework } from "@superfluid-finance/sdk-core";
import { ethers } from "ethers";
```

```
const provider = new ethers.providers.InfuraProvider(
  "matic",
  "<INFURA_API_KEY>"
);

const sf = await Framework.create({
  chainId: 137, //this is for matic - enter your own chainId here
  provider
});

// create a signer
const signer = sf.createSigner({ privateKey: "<TEST_ACCOUNT_PRIVATE_KEY>", provider });

// load the usdcx SuperToken via the Framework
const usdcx = sf.loadSuperToken("0xCAa7349CEA390F89641fe306D93591f87595dc1F");

// create an approve operation
const approveOperation = usdcx.approve({ receiver: "0xab...", amount: ethers.utils.pars

// execute the approve operation, passing in a signer
const txn = await approveOperation.exec(signer);

// wait for the transaction to be confirmed
const receipt = await txn.wait();

// or you can create and execute the transaction in a single line
const approveTxn = await usdcx.approve({ receiver: "0xab...", amount: ethers.utils.pars
const approveTxnReceipt = await approveTxn.wait();
```

```
// load the usdcx SuperToken via the Framework
const usdcx = sf.loadSuperToken("0xCAa7349CEA390F89641fe306D93591f87595dc1F");

// create an approve operation
const approveOperation = usdcx.approve({ receiver: "0xab...", amount: ethers.utils.pars

// execute the approve operation, passing in a signer
const txn = await approveOperation.exec(signer);

// wait for the transaction to be confirmed
const receipt = await txn.wait();

// or you can create and execute the transaction in a single line
const approveTxn = await usdcx.approve({ receiver: "0xab...", amount: ethers.utils.pars
const approveTxnReceipt = await approveTxn.wait();
```


Developers - Previous
SDK Core General

Next
Functionality

Functionality

Each Superfluid Developer Subject (Super Tokens, CFA, etc.) has a page dedicated to relevant usage of the SDK Core. Find them listed out below 👇

Super Tokens

Super Tokens - SDK Core

Constant Flow Agreement (CFA)

SDK Core

Access Control List for CFA

ACL Features - SDK

Instant Distribution Agreement (IDA)

IDA - SDK Core

Batch Calls

Batch Calls - SDK Core

Previous
Initialization

Next
Getting Data

Last modified 10mo ago

ACL Features - SDK

Instant Distribution Agreement (IDA)

IDA - SDK Core

Batch Calls

Batch Calls - SDK Core

Previous
Initialization

Next
Getting Data

Last modified 10mo ago

Getting Data

How to use the SDK Core to write subgraph queries and get account data

When working on a Superfluid related application, you'll likely need to use the graph to query data about what's happening with your users & Superfluid agreements.

Luckily, the SDK-Core has functionality which will allow you to easily query the graph to get data about accounts & their interaction with the Superfluid protocol. You can also use the CFAV1 & IDAV1 methods to get data that is specific to those agreements. In this section, we'll show allow you to get this data. If you'd like to go deeper on the topic, we'd recommend reading through our section on the Superfluid Subgraph.

Pre-Defined Subgraph Queries

Here is a list of pre defined queries you can use via the sdk. Once the framework is initialized, you can use the pattern modeled below to run each query (i.e. `sf.query.INSERT_QUERY`).

```
const { Framework } = require("@superfluid-finance/sdk-core");
const { ethers } = require("ethers");

const provider = new ethers.providers.InfuraProvider(
```

```
  "matic",
  "<INFURA_API_KEY>"
);
const sf = await Framework.create({
  networkName: "matic",
  provider
});

type Paging = { take: number, skip?: number, lastId?: string };

const pageResult = await sf.query.
  // The different queries can take different order by properties
  // given the properties that exist on the entity itself.
  listAllSuperTokens({ isListed?: boolean },
    paging: Paging,
    ordering: Ordering<Token_OrderBy>
  );

  listIndexes({ indexId?: string, publisher?: string, token?: string },
    paging: Paging,
    ordering: Ordering<Index_OrderBy>
  );

  listIndexSubscriptions({ subscriber?: string, approved?: boolean },
    paging: Paging,
    ordering: Ordering<IndexSubscription_OrderBy>
  );
```

```
listStreams({ sender?: string, receiver?: string, token?: string },
  paging: Paging,
  ordering: Ordering<Stream_OrderBy>
);

listUserInteractedSuperTokens({ account?: string, token?: string },
  paging: Paging,
  ordering: Ordering<AccountTokenSnapshot_OrderBy>
);

listEvents({ account?: string, timestamp_gt?: number },
  paging: Paging,
  ordering: Ordering<Event_OrderBy>
);

// A subscription function which allows you to subscribe to events via polling.
on(
      callback: (events: AllEvents[], unsubscribe: () => void) => void,
      ms: number,
      account?: string,
      timeout?: number
  )
```

Direct Initialization

If you'd like, you can also initialize the `Query` class as a standalone class like so:

If you'd like, you can also initialize the `Query` class as a standalone class like so:

```
import { Query } from "@superfluid-finance/sdk-core";
const query = new Query({
  customSubgraphQueriesEndpoint: "<A_CUSTOM_ENDPOINT>",
  dataMode: "SUBGRAPH_ONLY" | "SUBGRAPH_WEB3" | "WEB3_ONLY"
});
```

Pagination

All of the pre-defined query functions will accept pagination options: `({ skip: number, take: number })`, if you don't pass anything in, it will use a default of: `{ skip: 0, take: 100 }`. You can also paginate by `lastId`, this allows you to bypass the limitation of the max skip of 5000 entities.

> **Note**: this example uses the `graphql-request` library, but you just need to provide a valid query which is a string.

Ordering

You can also pass in an ordering object for the different queries, each query function will accept different ordering properties depending on the properties on the entity. We have different defaults for each so you don't need to actually pass anything in.

Example Usage:

```
const { Framework } = require("@superfluid-finance/sdk-core");
const { ethers } = require("ethers");

const provider = new ethers.providers.InfuraProvider(
        "matic",
        "<INFURA_API_KEY>"
);
const sf = await Framework.create({
  networkName: "matic",
        provider
});
const results = await sf.query.listAllSuperTokens(
  { isListed: true },
  { skip: 5, take: 150 },
  {
    orderBy: "createdAtBlockNumber",
    orderDirection: "desc"
  });
```

Getting Data from Agreements

CFAV1 Read Operations

```
// Read functions
await sf.cfaV1.getFlow({
  superToken: string,
  sender: string,
  receiver: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await sf.cfaV1.getAccountFlowInfo({
  superToken: string,
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await sf.cfaV1.getNetFlow({
  superToken: string,
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

IDAV1 Read Operations

```
// Read functions
await sf.idaV1.getSubscription({
  superToken: string,
```

```
  publisher: string,
  indexId: string,
  subscriber: string,
  providerOrSigner: string
});

await sf.idaV1.getIndex({
  superToken: string,
  publisher: string,
  indexId: string,
  providerOrSigner: string
});
```

Super Token Read Operations

```
// Read functions
await usdcx.balanceOf({
  account: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.allowance({
  owner: string,
  spender: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
```

```
await usdcx.allowance({
  owner: string,
  spender: string,
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.name({
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.symbol({
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});

await usdcx.totalSupply({
  providerOrSigner: ethers.providers.Provider | ethers.Signer
});
```

← Previous
Functionality

Next
Resolver →

Tracking Super Token Balances

Super Token balances can change every second! How do we approach tracking them?

Compatibility

As Super Tokens are ERC20, they are, generally speaking, backward compatible with the Ethereum infrastructure and ecosystem. You can see your balance in Metamask, transfer funds using Gnosis Safe / swap Super Tokens on Uniswap. Forward compatibility is a bit trickier. Gnosis Safe and Metamask are able to display your balance correctly when you are receiving streams, but that doesn't mean you can "stream-swap" on Uniswap! (at least not yet anyways 😉).

Balance Tracking Considerations

Some applications (notably Etherscan) rely on `transfer` __ events to first start tracking a user's balance. For scalability reasons however, agreements can't emit `transfer` events every time a user's balance changes. This leads to certain apps not showing all of a SuperToken's holders, and on occasion showing incorrect balances.

In order to work around this, applications looking to support Super Tokens should look not only to

user's balance changes. This leads to certain apps not showing all of a SuperToken's holders, and on occasion showing incorrect balances.

In order to work around this, applications looking to support Super Tokens should look not only to `transfer` events, but also to Superfluid's `AgreementStateUpdated` event. This is triggered every time an account's state changes, which is likely to indicate a future change in their balance.

Once you know you need to track an account, the best way to get an up to date balance is by calling the `balanceOf` function. As you know by now, Superfluid enables a number of new ways to move funds. Only tracking transfers won't cut it!

An excellent open source tool for checking user balances in real time is the Superfluid Console. We suggest leveraging it for your own usage in the protocol:

Superfluid Console

← Previous
In-Depth

Next
ERC777 in Super Tokens →

Last modified 10mo ago

Resolver

This helper contract makes it easy to find the contract addresses for the protocol

The Resolver is a helper contract which allows you to quickly find all the contract addresses for the Superfluid Framework. You can think of it as on-chain Superfluid directory.

The resolver is always deployed when the protocol is deployed. You've probably used it already and didn't know it, as this is what the `@superfluid-finance/sdk-core` uses each time you call `Framework.create()`.

If you're using the SDK Core, it's unlikely that you need the resolver, because the SDK exposes various methods which call the resolver behind the scenes to access specific contract addresses. For example, if you need to access a Super Token, you can do so by symbol using:

```
await sf.loadSuperToken("USDCx");
```

Similarly, if you need the address of any specific agreement, you can access these addresses from the Framework object itself. For example, if the Framework object is instantiated for the Goerli network like so:

```
const sf = await Framework.create({
```

```
    networkName: "goerli",
    provider
});
```

You can access the address of the host contract using the Framework object:

`sf.settings.config.hostAddress`

And the addresses of the constant flow agreement or instant distribution agreement contracts like so:

```
sf.settings.config.cfaV1Address
sf.settings.config.idaV1Address
```

> These addresses are also accessible in the Network Directory and with the Superfluid Console.

Accessing Contracts In Solidity with the Host and Resolver

Accessing these contracts in solidity is a bit different. However, it can be just as easy if you know how to use the Superfluid Resolver and/or the Superfluid Host contract to get these addresses. In the below contract, you'll see 3 sample functions which demonstrate how to access the addresses of already deployed contracts:

```
pragma solidity 0.8.13;
```

```
import { IResolver } from "@superfluid-finance/ethereum-contracts/contracts/interfaces/u
import { ISuperfluid } from "@superfluid-finance/ethereum-contracts/contracts/interfaces

contract ResolverExample {
    IResolver public resolver;
    ISuperfluid public host;

    constructor(IResolver _resolver, ISuperfluid _host) {
        resolver = _resolver;
        host = _host;
    }

    function getContractFromResolver(string calldata name) external view returns (addres
        address addr = resolver.get(name);
        return addr;
    }

    function getCFAContract() external view returns (address) {
        bytes32 agreement;
        agreement = keccak256("org.superfluid-finance.agreements.ConstantFlowAgreement.v

        address agreementAddress = address(host.getAgreementClass(agreement));
        return agreementAddress;
    }

    function getIDAContract() external view returns (address) {
```

```
        bytes32 agreement;
        agreement = keccak256("org.superfluid-finance.agreements.InstantDistributionAgre

        address aggrementAddress = address(host.getAgreementClass(agreement));
        return agreementAddress;
    }
}
```

To get the address of a Superfluid agreement like the Constant Flow Agreement or Instant Distribution agreement, you can use the Superfluid Host contract's `getAgreementClass` method. You can see this being done in both the `getCFAContract()` and `getIDAContract()` examples in the above contract. You need to hash the following string:

```
`org.superfluid-finance.agreements.${agreementName}.v1`
```

Then pass it as an argument to the `host.getAgreementClass` method. From here, you need to get the address of the returned contract using the same syntax seen in the above contract on lines 24 and 32.

In the case of other contracts, you can use the Resolver. If you look deep into the Superfluid SDK Core, you'll see that the resolver makes use of the Superfluid Loader contract to load in the entire Framework.

To get a specific token address using the Resolver, you can use the `resolver.get()` method as seen in the above contract with the following scheme:

seen in the above contract with the following schema:

```
//for Super Tokens
resolver.get(`supertokens.${protocolReleaseVersion}.superTokenSymbol`);

//example
resolver.get('supertokens.v1.DAIx');
```

```
//for ERC20 tokens listed with the protocol
resolver.get(`tokens.tokensymbol`);

//example
resolver.get('tokens.DAI');
```

If you'd like, you can also use the Resolver to get the entirety of the framework with the Superfluid Loader. You need to pass in the following argument to the `resolver.get()` function to make this happen:

```
resolver.get('SuperfluidLoader-v1');
```

The above line of code will return the address of the Superfluid Loader contract on your current network. This method will return an array which contains 4 of the key addresses used within the protocol: the host, super token factory, CFA, and IDA contract. The function needed to do this is the `loadFramework` function, with the current protocol version passed in as a lone string parameter:

```
resolver.get('tokens.DAI');
```

If you'd like, you can also use the Resolver to get the entirety of the framework with the Superfluid Loader. You need to pass in the following argument to the `resolver.get()` function to make this happen:

```
resolver.get('SuperfluidLoader-v1');
```

The above line of code will return the address of the Superfluid Loader contract on your current network. This method will return an array which contains 4 of the key addresses used within the protocol: the host, super token factory, CFA, and IDA contract. The function needed to do this is the `loadFramework` function, with the current protocol version passed in as a lone string parameter:

```
superfluidLoader.loadFramework("v1");
```

← Previous
Getting Data

Next - Developers
Solidity General →

Solidity General

Working with the CFA & IDA in Solidity - and using the CFA Library

This section will help you to interact with Superfluid at the smart contract level.

> NOTE: we recommend using the Superfluid Console and the Superfluid Dashboard as you build on Superfluid.

Installation

```
npm install @superfluid-finance/ethereum-contracts
```

Solidity x Superfluid Overview

Calling Super Agreements

Superfluid Solidity Helper Libraries

Solidity

IDA - Solidity

Access Control List

Access Control List (ACL)

Deploying the Framework

Deploying The Framework

User Data

Access Control List (ACL)

Deploying the Framework

Deploying The Framework

User Data

User Data

Previous
Resolver

Next
Calling Super Agreements

Last modified 6mo ago

Calling Super Agreements

Using Superfluid at the Smart Contract Level

Working With Superfluid Using Solidity

When you open a stream or make use of instant distributions, what you're really doing is interacting with Superfluid *agreements*. Money streams are created by interacting with the constant flow agreement (CFA), while calls related to instant distributions are done by working with the instant distribution agreement (IDA).

> NOTE: the easiest way to create money streams or work with the IDA in solidity is via the CFAv1 Library and IDAv1 Library.

To get started, you'll need to be sure to import, at minimum, the Superfluid host interface & agreement interface that you'd like to work with. In a situation where you'd like to use both the CFA and IDA in the same contract, you would import these contracts like this:

```
pragma solidity ^0.8.0
```

```
import { ISuperfluid }from "@superfluid-finance/ethereum-contracts/contracts/interfaces/

import { IConstantFlowAgreementV1 } from "@superfluid-finance/ethereum-contracts/contrac

import { IInstantDistributionAgreementV1 } from "@superfluid-finance/ethereum-contracts/
```

If you'd like to interact with a Superfluid agreement directly by using solidity, you need to make a function call first to the Superfluid *host* contract. You'll call the `callAgreement` or `callAgreementWithContext` function on the host contract (`Superfluid.sol`), and pass in a few parameters:

`ISuperAgreement agreementClass` - the the agreement you're going to interact with (either the CFA or IDA)

`bytes memory calldata` - the transaction you're calling on the agreement you're interacting with, compiled to bytecode (using one of solidity's encoding methods)

`bytes memory userData` - any additional data you'd like to include with your function call. If you don't plan to add userData, you can pass in an empty bytes value (i.e. `"0x"`). You can learn more about this parameter here.

Here's an example of how this looks in action when interacting with the constant flow agreement. This pattern will be the same whether you're creating, updating, or deleting flows.

```
//creating a flow in pure solidity
host.callAgreement(
```

```
host.callAgreement(
    cfa,
    abi.encodeWithSelector(
        cfa.createFlow.selector,
        token,
        receiver,
        flowRate,
        new bytes(0) // placeholder - always pass in bytes(0)
    ),
    "0x" //userData
);
```

The following is an example for interacting with the instant distribution agreement using solidity. This pattern is the same for each interaction you'd like to make with the IDA:

```
// distributing tokens with the instant distribution agreement
host.callAgreement(
    ida,
    abi.encodeWithSelector(
        ida.distribute.selector,
        token,
        index,
        amountToDistribute,
        new bytes(0) // placeholder ctx
    ),
    "0x" // user data
);
```

> NOTE: If you're interacting with agreements inside of a Super App callback, this process will work differently. See this section for details.

Common Mistakes

One other thing to keep in mind is the value of `msg.sender` when working with Superfluid from a contract. For example, developers will occasionally want to create a function which will let an account create a flow into the contract itself. So, they'll write a function that looks like this, expecting it to create a flow into the contract from the `msg.sender` of the function:

```
//this will fail because a contract cannot create a flow to itself

function createFlowFail(ISuperToken DAIx, int96 flowRate) external {
    cfaV1.createFlow(address(this), DAIx, flowRate);
}
```

In the above function, you have set the receiver on the `cfaV1.createFlow` function to be the contract's address. Under the hood, even though `msg.sender` on the broader `createFlowFail` function is an external address, the msg.sender on the `cfa.createFlow` call to the Superfluid protocol is the address of the *contract*. The Superfluid `callAgreement` function will see that the contract is trying to create a flow into itself, and revert.

If you want to create a flow into a contract, we recommend that you use either the Core SDK from

If you want to create a flow into a contract, we recommend that you use either the Core-SDK from outside of a contract or a 2nd contract which already has some balance of Super Tokens.

A similar mistake which stems from this `msg.sender` misunderstanding occurs when developers want to allow external accounts to create flows directly to other accounts via a function on the contract. For example:

```
//remember that this will revert if the contract has a zero balance of DAIx
//msg.sender on the call to the Superfluid framework is the contract

function createFlow2Receiver(address receiver, ISuperToken DAIx, int96 flowRate) extern
    cfaV1.createFlow(receiver, DAIx, flowRate);
}
```

Recall that the value of `msg.sender` on the `cfaV1.createFlow` function will be the contract's address, not the external address calling the broader `createFlow2Receiver` function. If you're intending to create a flow from the contract to another address, then this code is what you need (just make sure your contract has a balance of Super tokens!). However, if you're running this with the expectation that it will open a flow directly from an external account calling the function to the intended receiver, it will not work as expected.

← Developers - Previous
Solidity General

Next
Libraries →

contract. For example:

```
//remember that this will revert if the contract has a zero balance of DAIx
//msg.sender on the call to the Superfluid framework is the contract

function createFlow2Receiver(address receiver, ISuperToken DAIx, int96 flowRate) extern
    cfaV1.createFlow(receiver, DAIx, flowRate);
}
```

Recall that the value of `msg.sender` on the `cfaV1.createFlow` function will be the contract's address, not the external address calling the broader `createFlow2Receiver` function. If you're intending to create a flow from the contract to another address, then this code is what you need (just make sure your contract has a balance of Super tokens!). However, if you're running this with the expectation that it will open a flow directly from an external account calling the function to the intended receiver, it will not work as expected.

← Developers - Previous
Solidity General

Next
Libraries →

Libraries

CFAv1 Library

Simplify money streaming in Solidity

Solidity

IDAv1 Library

Simplify performing instant distributions in Solidity

IDA - Solidity

Previous

Next

CFAv1 Library

Simplify money streaming in Solidity

Solidity

IDAv1 Library

Simplify performing instant distributions in Solidity

IDA - Solidity

Previous
Calling Super Agreements

Next
Resolver

Last modified 10mo ago

Resolver

This helper contract makes it easy to find the contract addresses for the protocol

The Resolver is a helper contract which allows you to quickly find all the contract addresses for the Superfluid Framework. You can think of it as on-chain Superfluid directory.

The resolver is always deployed when the protocol is deployed. You've probably used it already and didn't know it, as this is what the `@superfluid-finance/sdk-core` uses each time you call `Framework.create()`.

If you're using the SDK Core, it's unlikely that you need the resolver, because the SDK exposes various methods which call the resolver behind the scenes to access specific contract addresses. For example, if you need to access a Super Token, you can do so by symbol using:

```
await sf.loadSuperToken("USDCx");
```

Similarly, if you need the address of any specific agreement, you can access these addresses from the Framework object itself. For example, if the Framework object is instantiated for the Goerli network like so:

```
const sf = await Framework.create({
```

```
    networkName: "goerli",
    provider
});
```

You can access the address of the host contract using the Framework object:

`sf.settings.config.hostAddress`

And the addresses of the constant flow agreement or instant distribution agreement contracts like so:

```
sf.settings.config.cfaV1Address
sf.settings.config.idaV1Address
```

> These addresses are also accessible in the Network Directory and with the Superfluid Console.

Accessing Contracts In Solidity with the Host and Resolver

Accessing these contracts in solidity is a bit different. However, it can be just as easy if you know how to use the Superfluid Resolver and/or the Superfluid Host contract to get these addresses. In the below contract, you'll see 3 sample functions which demonstrate how to access the addresses of already deployed contracts:

```
pragma solidity 0.8.13;
```

```
import { IResolver } from "@superfluid-finance/ethereum-contracts/contracts/interfaces/u
import { ISuperfluid } from "@superfluid-finance/ethereum-contracts/contracts/interfaces

contract ResolverExample {
    IResolver public resolver;
    ISuperfluid public host;

    constructor(IResolver _resolver, ISuperfluid _host) {
        resolver = _resolver;
        host = _host;
    }

    function getContractFromResolver(string calldata name) external view returns (addres
        address addr = resolver.get(name);
        return addr;
    }

    function getCFAContract() external view returns (address) {
        bytes32 agreement;
        agreement = keccak256("org.superfluid-finance.agreements.ConstantFlowAgreement.v

        address agreementAddress = address(host.getAgreementClass(agreement));
        return agreementAddress;
    }

    function getIDAContract() external view returns (address) {
```

```
        bytes32 agreement;
        agreement = keccak256("org.superfluid-finance.agreements.InstantDistributionAgre

        address aggrementAddress = address(host.getAgreementClass(agreement));
        return agreementAddress;
    }
}
```

To get the address of a Superfluid agreement like the Constant Flow Agreement or Instant Distribution agreement, you can use the Superfluid Host contract's `getAgreementClass` method. You can see this being done in both the `getCFAContract()` and `getIDAContract()` examples in the above contract. You need to hash the following string:

```
`org.superfluid-finance.agreements.${agreementName}.v1`
```

Then pass it as an argument to the `host.getAgreementClass` method. From here, you need to get the address of the returned contract using the same syntax seen in the above contract on lines 24 and 32.

In the case of other contracts, you can use the Resolver. If you look deep into the Superfluid SDK Core, you'll see that the resolver makes use of the Superfluid Loader contract to load in the entire Framework.

To get a specific token address using the Resolver, you can use the `resolver.get()` method as seen in the above contract with the following schema:

seen in the above contract with the following schema:

```
//for Super Tokens
resolver.get(`supertokens.${protocolReleaseVersion}.superTokenSymbol`);

//example
resolver.get('supertokens.v1.DAIx');
```

```
//for ERC20 tokens listed with the protocol
resolver.get(`tokens.tokensymbol`);

//example
resolver.get('tokens.DAI');
```

If you'd like, you can also use the Resolver to get the entirety of the framework with the Superfluid Loader. You need to pass in the following argument to the `resolver.get()` function to make this happen:

```
resolver.get('SuperfluidLoader-v1');
```

The above line of code will return the address of the Superfluid Loader contract on your current network. This method will return an array which contains 4 of the key addresses used within the protocol: the host, super token factory, CFA, and IDA contract. The function needed to do this is the `loadFramework` function, with the current protocol version passed in as a lone string parameter:

```
resolver.get('tokens.DAI');
```

If you'd like, you can also use the Resolver to get the entirety of the framework with the Superfluid Loader. You need to pass in the following argument to the `resolver.get()` function to make this happen:

```
resolver.get('SuperfluidLoader-v1');
```

The above line of code will return the address of the Superfluid Loader contract on your current network. This method will return an array which contains 4 of the key addresses used within the protocol: the host, super token factory, CFA, and IDA contract. The function needed to do this is the `loadFramework` function, with the current protocol version passed in as a lone string parameter:

```
superfluidLoader.loadFramework("v1");
```

← Previous
Libraries

Next - Developers
Testing Guide →

Testing Guide

How to test with Superfluid actions in Foundry & Hardhat

Learn how to **deploy the Superfluid Framework locally** and **mint fake Super Tokens** to get set with your testing.

> ⓘ 💡 These examples reference the MoneyRouter contract which is covered in detail in this video on money streaming in solidity.

But what about truffle/brownie/other environments?

We have a legacy truffle example that you can find here. It is very similar to the Hardhat example above, but with some minor differences in terms of imports & scripts used for framework deployment.

A brownie/python example is in the works. Stay tuned (:

If you have any questions on how to set up your tests for a Superfluid project, please contact us in Discord.

But what about truffle/brownie/other environments?

We have a legacy truffle example that you can find here. It is very similar to the Hardhat example above, but with some minor differences in terms of imports & scripts used for framework deployment.

A brownie/python example is in the works. Stay tuned (:

If you have any questions on how to set up your tests for a Superfluid project, please contact us in Discord.

Join the Superfluid Discord Server!
Discord

← Previous
Resolver

Next
Hardhat Testing →

Last modified 8mo ago

Hardhat Testing

Video Tutorial

For the visually inclined, view our video explainer **HERE**.

Example Code

super-examples/MoneyRouter.test.js at main · superfluid-finance/super-examples
GitHub

Hardhat test suite from our Money Router example

Hardhat Example

We recommend including the following imports and deployment scripts when setting up your hardhat tests. The `deployTestFramework()` script will allow you to deploy the framework using the SuperfluidFrameworkDeployer contract, and to call `deployWrapperSuperToken()` to mint fake Super Tokens for your tests.

Dependencies you'll need are listed at the top of the file, but if you already have a hardhat project

set up, you'll likely only need `@superfluid-finance/sdk-core` and `@superfluid-finance/ethereum-contracts`.

```
const { expect } = require("chai")
const { Framework } = require("@superfluid-finance/sdk-core")
const { ethers } = require("hardhat")
const { deployTestFramework } = require("@superfluid-finance/ethereum-contracts/dev-scr
const TestToken = require("@superfluid-finance/ethereum-contracts/build/contracts/TestT

let sfDeployer
let contractsFramework
let sf
let moneyRouter
let dai
let daix

// Test Accounts
let owner
let account1
let account2

const thousandEther = ethers.utils.parseEther("10000")

before(async function () {

    // get hardhat accounts
    [owner, account1, account2] = await ethers.getSigners();
```

```
sfDeployer = await deployTestFramework();

// GETTING SUPERFLUID FRAMEWORK SET UP

// deploy the framework locally
contractsFramework = await sfDeployer.frameworkDeployer.getFramework()

// initialize framework
sf = await Framework.create({
    chainId: 31337,
    provider: owner.provider,
    resolverAddress: contractsFramework.resolver, // (empty)
    protocolReleaseVersion: "test"
})

// // DEPLOYING DAI and DAI wrapper super token (which will be our `spreaderToken`)
tokenDeployment = await sfDeployer.frameworkDeployer.deployWrapperSuperToken(
    "Fake DAI Token",
    "fDAI",
    18,
    ethers.utils.parseEther("100000000").toString()
);

// DEPLOYING DAI and DAI wrapper super token (which will be our `spreaderToken`)
daix = await sf.loadSuperToken("fDAIx")
dai = new ethers.Contract(
    daix.underlyingToken.address,
```

```
    TestToken.abi,
    owner
)
// minting test DAI
await dai.mint(owner.address, thousandEther)
await dai.mint(account1.address, thousandEther)
await dai.mint(account2.address, thousandEther)

// approving DAIx to spend DAI (Super Token object is not an ethers contract object
await dai.approve(daix.address, ethers.constants.MaxInt256)
await dai
    .connect(account1)
    .approve(daix.address, ethers.constants.MaxInt256)
await dai
    .connect(account2)
    .approve(daix.address, ethers.constants.MaxInt256)
// Upgrading all DAI to DAIx
const ownerUpgrade = daix.upgrade({amount: thousandEther});
const account1Upgrade = daix.upgrade({amount: thousandEther});
const account2Upgrade = daix.upgrade({amount: thousandEther});

await ownerUpgrade.exec(owner)
await account1Upgrade.exec(account1)
await account2Upgrade.exec(account2)

let MoneyRouter = await ethers.getContractFactory("MoneyRouter", owner)

moneyRouter = await MoneyRouter.deploy(
```

```
    const account2Upgrade = daix.upgrade({amount: thousandEther});

    await ownerUpgrade.exec(owner)
    await account1Upgrade.exec(account1)
    await account2Upgrade.exec(account2)

    let MoneyRouter = await ethers.getContractFactory("MoneyRouter", owner)

    moneyRouter = await MoneyRouter.deploy(
        owner.address
    )
    await moneyRouter.deployed()
});


//Write your tests...
```


Developers - Previous
Testing Guide

Next
Foundry Testing

Foundry Testing

Video Tutorial

For the visually inclined, view our video explainer **HERE**.

Example Code

super-examples/MoneyRouter.t.sol at main · superfluid-finance/super-examples
GitHub

Foundry test suite from our Money Router example

Foundry Example

Within a foundry test file, you can call the SuperfluidFrameworkDeployer contract directly in your `setUp()` function.

```
import "forge-std/Test.sol";
import "../../contracts/MoneyRouter.sol";
import {ISuperfluid} from "@superfluid-finance/ethereum-contracts/contracts/interfaces/s
```

```
import {
    SuperfluidFrameworkDeployer,
    TestGovernance,
    Superfluid,
    ConstantFlowAgreementV1,
    CFAv1Library,
    SuperTokenFactory
} from "@superfluid-finance/ethereum-contracts/contracts/utils/SuperfluidFrameworkDeplo

contract StreamRebounderRandomTest is Test {

    StreamRebounder public streamRebounder;

    struct Framework {
        TestGovernance governance;
        Superfluid host;
        ConstantFlowAgreementV1 cfa;
        CFAv1Library.InitData cfaLib;
        InstantDistributionAgreementV1 ida;
        IDAv1Library.InitData idaLib;
        SuperTokenFactory superTokenFactory;
    }

    SuperfluidFrameworkDeployer.Framework sf;

    function setUp() public {
```

```
        address public owner;
        //DEPLOYING THE FRAMEWORK
        SuperfluidFrameworkDeployer sfDeployer = new SuperfluidFrameworkDeployer();
        sf = sfDeployer.getFramework();

        // DEPLOYING DAI and DAI wrapper super token

        vm.prank(owner);
        ISuperToken daix = sfDeployer.deployWrapperToken(
            "Fake DAI", "DAI", 18, 10000000000000
        );

        MoneyRouter = new StreamRebounder(
            sf.host,
            owner
        );
    }

    //add other functions and test contracts...

}
```

When using foundry, you'll need to install `superfluid-finance/ethereum-contracts`. If you have issues when running `forge install`, try using the `--no-commit` flag.

```
        ISuperToken daix = sfDeployer.deployWrapperToken(
            "Fake DAI", "DAI", 18, 10000000000000
        );

        MoneyRouter = new StreamRebounder(
            sf.host,
            owner
        );
    }

    //add other functions and test contracts...

}
```

When using foundry, you'll need to install `superfluid-finance/ethereum-contracts`. If you have issues when running `forge install`, try using the `--no-commit` flag.

← Previous
Hardhat Testing

Next
In-Depth →

In-Depth

Learn how to prepare a Hardhat test script for a fork that lets you test with Superfluid

Hardhat Mainnet Fork Testing

Learn how to simulate Super App registration so you can test your Super Apps on forks. Demo'd in Hardhat as well.

Super App Testing on Mainnet Forks

Previous
Foundry Testing

Next
Hardhat Mainnet Fork Testing

Last modified 8mo ago

In-Depth

Learn how to prepare a Hardhat test script for a fork that lets you test with Superfluid

Hardhat Mainnet Fork Testing

Learn how to simulate Super App registration so you can test your Super Apps on forks. Demo'd in Hardhat as well.

Super App Testing on Mainnet Forks

Previous
Foundry Testing

Next
Hardhat Mainnet Fork Testing

Last modified 8mo ago

Hardhat Mainnet Fork Testing

Set yourself to easily test with Superfluid on a Hardhat fork

All that's needed here is to set up your SDK Core Framework object and you'll be on your way to easily simulating mainnet Superfluid actions the way you would normally.

First, make sure you have these required imports and make yourself an `sf` global variable for your Framework object.

```
const { Framework } = require("@superfluid-finance/sdk-core");
const { ethers } = require("hardhat");

// Superfluid Global Variable
let sf;
```

Then, create the Framework object, finding the Resolver address from your desired chain on the Contract Addresses page.

```
before(async function () {

    sf = await Framework.create({
```

```
sf = await Framework.create({
    provider: ethers.provider,
    resolverAddress: <<resolver address from your desired chain>>,
    networkName: "hardhat",
    dataMode: "WEB3_ONLY",
    protocolReleaseVersion: "v1",
    chainId: 31337
});

...
```

Now You're All Set!

Notice that there was no need to deploy the Framework is already exists on the chain you are testing.

← Previous
In-Depth

Next
Super App Testing on Mainnet Forks →

Last modified 8mo ago

```
sf = await Framework.create({
    provider: ethers.provider,
    resolverAddress: <<resolver address from your desired chain>>,
    networkName: "hardhat",
    dataMode: "WEB3_ONLY",
    protocolReleaseVersion: "v1",
    chainId: 31337
});

...
```

Now You're All Set!

Notice that there was no need to deploy the Framework is already exists on the chain you are testing.

← Previous
In-Depth

Next
Super App Testing on Mainnet Forks →

Last modified 8mo ago