Lesson Type: Practical | Estimated Time: 2-4 hours | Current Score: 100%

Build sovereign user profiles using Ceramic Self.ID

Ceramic is a decentralized data network, that allows for building composable Web3 applications. Since Ceramic decentralizes application databases, application developers can reuse data across applications and make them automatically interoperable.

In the course of this article, we will dig deeper into what those statements actually mean, and why decentralized data is important in the first place.

Web3 and Data

In the past few years, we have seen Web3 trends like DeFi, NFTs, and more recently DAOs blow up. Smart contract platforms like Ethereum have shown us that dApps which act as legos, and can be composed together with other dApps to build entirely new dApps have a lot of value. This is particularly highlighted with tokens that build on top of other tokens, DeFi protocols which utilize other DeFi protocols, etc.

Ceramic allows you to bring that same type of composability to data on the internet. This can be any kind of data. Profiles, social connections, blog posts, identities, reputation, game assets, etc.

This is an abstract concept, and it is a bit tricky to try and define Ceramic Network as a single thing. Similar to Ethereum, which by itself is hard to define (what does it even mean to be a smart contract plaform?), it is easier to understand if we look at specific use cases that Ceramic can enable and examples of a vision for the future. Ethereum is much easier to understand when we look at exact examples like DeFi protocols, NFTs, DAOs etc - the same thinking can be applied while attempting to understand Ceramic.

Lack of data in Web3

The web3 market right now is mainly comprised of financial applications. Things to do with tokens and digital assets. This is somewhat due to design. Smart contracts are inherently limited in how much data can be stored in them, and there's only so much data functionality that can be built.

As we progress and dApps mature, the market demand for building more data-rich applications is increasing. Trends like decentralized social medias, decentralized reputation platforms, decentralized blogs, etc. is taking off, with a lot of technical approaches being taken, but lacking in one aspect or the other.

In Web2, these platforms were built as data silos. Your Twitter posts and social connections are locked into the Twitter platform - you cannot transfer your social connections on Twitter to Facebook, you cannot transfer your Twitter posts to Facebook, etc. They are all built as silos.

This will change. Ceramic is betting on interoperable applications and interoperable ecosystems being able to outcompete these siloed platforms.

So... What does Ceramic do?

Ceramic is building:

1. A generalized data protocol
2. where data can be modified only by the owner
3. with high volume data processing
4. with global data availability and consistency
5. with support for fast querying
6. with interoperable data across applications
7. and community governance

This is a lot. This is also why Ceramic can be tricky to define on it's own. After all, just like Ethereum, it is a generalized protocol, albeit for data.

To achieve the true scale and vision of Ceramic, a lot of breakthroughs need to be achieved. If Ceramic is to become the decentralized database of the web, it needs to be able to scale massively - more than any centralized database today as none of them are storing data from the entire internet.

Data also needs to be made globally available, and it needs to be ensured that Ceramic node runners make that data available for the rest of the world and don't hijack it.

Additionally, not just storage, Ceramic also needs to be fast to query and retrieve data from. Users typically read much more data than they write, so fast reads and queries are extremely important for Ceramic to work at scale.

All of this while maintaining a high level of security and privacy over data, and ensuring that it doesn't come crashing down one day.

It is helpful to look at specific use cases that can be enabled by Ceramic today and the benefits of having a mutable, decentralized, general purpose data protocol.

Ceramic Use Cases

Decentralized Reputation

Reputation is highly tied into a person's identity. On Twitter, it's followers and likes. On Instagram, it is the hearts. On Reddit, it's Karma, and on StackOverflow, it's points.

In the web3 ecosystem today, dApps can hardly do better than centralized reputation systems like the above examples, with each platform having it's own reputation system. This is largely because storing large amounts of decentralized data that can change over time was not viable. And even if Ethereum Layer 2's were to reduce storage costs massively, what about non-Ethereum chains? What happens to your reputation when you switch to NEAR or Flow or Solana?

Enter Ceramic.

dApps can use Ceramic's data protocol to build standardized multi-chain reputation systems. A user can connect multiple wallets to their decentralized identity, belong from different blockchains, and data can be written to and updated from the user's decentralized data store on Ceramic.

Therefore, regardless of what chain and dApp the user is using, they can carry around their reputation system with them.

Social Graphs

Similar to reputation, your social graph is also heavily centralized in today's world. Your Twitter followers, Facebook friends, LinkedIn connections, and Snapchat buddies are all differently siloed.

With censorship increasing on centralized social medias over time, and since social media companies are the biggest in the world with some of them having enough power to manipulate entire national elections, the need for decentralized social media has never been more.

However, if we are building decentralized social media, let's try to do better than the old systems. Instead of locking users into a platform, we can actually allow for interoperability and optionality.

dApps can follow standardized data models to store posts and social graphs. These social graphs are carried by the user to

whatever dApp they want to use. This means products compete on which offers the best experience, not who has the most vendor lock-in.

This also allows for smaller players and startups with better products to achieve easier market penetration. They can utilize the underlying data that already exists, and when users sign up on their platform all their data is carried along with them. For bigger players, there can be financial incentives for providing a large amount of data for that datamodel.

Multi-Wallet and Multi-Chain Identity

You could extrapolate the decentralized reputation use case to achieve generalized multi-wallet and multi-chain identity for users. Instead of tying up data in smart contracts or offchain based on wallet addresses, data can be tied up to a user's decentralized identity which can be controlled by multiple wallets across multiple chains.

This way, multi-chain dApps can have a decentralized, yet single, source of truth for a user's data.

Ceramic's Multi Chain Architecture

At the lowest level, there is a decentralized identity. The most common approach to decentralized identities on Ceramic is something called a **3ID** (Three ID) - named after the team behind Ceramic Network called 3Box Labs.

A user can link multiple wallets from multiple chains to a single 3ID. Currently, Ceramic supports more than 10 blockchains, and is continually adding support for more.

3IDs can own data on Ceramic Network. Data on Ceramic is referred to as **Streams.** Each stream therefore has an owner (or multiple owners).

Streams have unique **StreamIDs**, which remain the same over the lifetime of the stream. 3IDs can modify and update the contents of a Stream that they have ownership on.

Streams have a **genesis** state, which is the initial data the Stream was created with. Following the genesis state, users can create **commits** on Streams, which represent modifications to the data. The latest state of a Stream can be computed by starting from the genesis state and applying all the commits one by one. The latest state is also referred to as the **tip** of a stream.

Using Ceramic

Ceramic provides a suite of high-level and low-level libraries and SDK's to work with, depending on the use case.

For common use cases, developers can use the high-level SDK - Self.ID - which abstracts away most of the complexities of working with 3IDs and Streams.

For complex or customized use cases, developers can work with a lower-level Ceramic HTTP Client, which connects to a Ceramic node they can run on their own (or the public nodes), and manage 3IDs and Stream data manually.

For the purposes of this tutorial, we will stick with the Self.ID high-level SDK, as otherwise this tutorial will become extremely large. If you're interested in digging deeper, do check out the *Recommended Resources* section below.

Self.ID

Self.ID is a single, high level, library that encapsulates 3ID accounts, creating and setting up a 3ID, underlying calls to Ceramic nodes, all in a single package optimized to work in a browser.

The SDK also includes popular use cases like multi-chain user profiles built in, which makes it very easy for developers to retrieve and store multi-chain data linked to a 3ID.

BUIDL

We will create a simple Next.js application, which uses Self.ID. It will allow users to login to the website using their wallet of choice, which will be linked to their 3ID. Then, we will allow the user to write some data to their decentralized profile, and be able to retrieve it from the Ceramic Network.

For verification of this level, we will ask you to enter your profile's StreamID at the end.

Let's get started by creating a new `next` app. Run the following command to create a new Next.js application inside a folder named `ceramic-tutorial`

```
npx create-next-app@latest ceramic-tutorial
```

and press `Enter` for all the question prompts. This should create the `ceramic-tutorial` folder and setup the `next` app inside it. It will also initialize a git repository you can push to GitHub after making changes.

Let's now install the Self.ID npm packages, and a dependent library, to get started. From inside the `ceramic-tutorial` folder, run the following in your terminal

```
npm install @self.id/react @self.id/web key-did-provider-ed25519
```

Let's also install the `ethers` and `web3modal` packages that we will be using to support wallet connections

```
npm install ethers web3modal
```

Open up the `ceramic-tutorial` folder in your text editor of choice, and let's get to coding.

The first thing we need to do is add Self.ID's Provider to the application. The SDK exposes a `Provider` component that needs to be added to the root of your web app. This initiatalizes the Self.ID instance, connects to the Ceramic Network, and makes Ceramic-related functionality available all across your app.

To do this, note in your `pages` folder Next.js automatically created a file called `_app.js`. This is the root of your web-app, and all other pages you create are rendered according to the configuration present in this file. By default, it does nothing special, and just renders your page directly. In our case, we want to wrap every component of ours with the Self.ID provider.

First, let's import the `Provider` from Self.ID. Add the following line at the top of `_app.js`

```
import { Provider } from "@self.id/react";
```

Next, change the `MyApp` function in that file to return the `Component` wrapped inside a `Provider`

```
function MyApp({ Component, pageProps }) {
  return (
    <Provider client={{ ceramic: "testnet-clay" }}>
      <Component {...pageProps} />;
    </Provider>
  );
}
```

We also specified a configuration option for the Provider. Specifically, we said that we want the Provider to connect to the Clay Test Network for Ceramic.

Let's make sure everything is working fine so far. Run the following in your terminal

```
npm run dev
```

Your website should be up and running at `http://localhost:3000`.

Lastly, before we start writing code for the actual application, replace the CSS in `styles/Home.module.css` with the following:

```
.main {
  min-height: 100vh;
}
```

```css
.navbar {
  height: 10%;
  width: 100%;
  display: flex;
  flex-direction: row;
  justify-content: space-between;
  padding-top: 1%;
  padding-bottom: 1%;
  padding-left: 2%;
  padding-right: 2%;
  background-color: orange;
}

.content {
  height: 80%;
  width: 100%;
  padding-left: 5%;
  padding-right: 5%;
  display: flex;
  flex-direction: column;
  align-items: center;
}

.flexCol {
  display: flex;
  flex-direction: column;
  align-items: center;
}

.connection {
  margin-top: 2%;
}

.mt2 {
  margin-top: 2%;
}

.subtitle {
  font-size: 20px;
  font-weight: 400;
}

.title {
  font-size: 24px;
  font-weight: 500;
}

.button {
  background-color: azure;
  padding: 0.5%;
  border-radius: 10%;
  border: 0px;
  font-size: 16px;
  cursor: pointer;
}

.button:hover {
  background-color: beige;
}
```

Since this is not a CSS tutorial, we will not be diving deep into these CSS properties, though you should be able to understand what these properties mean if you would like to dive deeper.

Alright, open up `pages/index.js` and remove everything currently present inside the `function Home() {...}`. We will replace the contents of that function with our code.

We will start by first initializing Web3Modal related code, as connecting a wallet is the first step.

Let's first import `Web3Provider` from `ethers`.

```
import { Web3Provider } from "@ethersproject/providers";
```

Then import these react hooks from `react` and `Web3Modal` from `web3modal`.

```
import { useEffect, useRef, useState } from "react";
import Web3Modal from "web3modal";
```

As before, let's create a reference using the `useRef` react hook to a Web3Modal instance in your `Home` function, and create a helper function to get the Provider.

```
const web3ModalRef = useRef();

const getProvider = async () => {
  const provider = await web3ModalRef.current.connect();
  const wrappedProvider = new Web3Provider(provider);
  return wrappedProvider;
};
```

This function will prompt the user to connect their Ethereum wallet, if not already connected, and then return a Web3Provider. However, if you try running it right now, it will fail because we have not yet initialized web3Modal.

Before we initialize Web3Modal, we will setup a React Hook provided to us by the Self.ID SDK. Self.ID provides a hook called `useViewerConnection` which gives us an easy way to connect and disconnect to the Ceramic Network. Add the following import

```
import { useViewerConnection } from "@self.id/react";
```

And now in your `Home` function, do the following to initialize the hook

```
const [connection, connect, disconnect] = useViewerConnection();
```

Now that we have this, we can initialize Web3Modal. Add the following `useEffect` React Hook to your `Home` function.

```
useEffect(() => {
  if (connection.status !== "connected") {
    web3ModalRef.current = new Web3Modal({
      network: "goerli",
      providerOptions: {},
      disableInjectedProvider: false,
    });
  }
}, [connection.status]);
```

We have seen this code before many times. The only difference is the conditional here. We are checking that if the user has not yet been connected to Ceramic, we are going to initialize the web3Modal.

The last thing we need to be able to connect to the Ceramic Network is something called an `EthereumAuthProvider`. It is a class exported by the Self.ID SDK which takes an Ethereum provider and an address as an argument, and uses it to connect your Ethereum wallet to your 3ID.

- To set that up, let us first import it

  ```
  import { EthereumAuthProvider } from "@self.id/web";
  ```

  and then add the following helper functions

  ```
  const connectToSelfID = async () => {
    const ethereumAuthProvider = await getEthereumAuthProvider();
    connect(ethereumAuthProvider);
  };

  const getEthereumAuthProvider = async () => {
    const wrappedProvider = await getProvider();
    const signer = wrappedProvider.getSigner();
    const address = await signer.getAddress();
    return new EthereumAuthProvider(wrappedProvider.provider, address);
  };
  ```

  `getEthereumAuthProvider` creates an instance of the `EthereumAuthProvider`. You may be wondering why we are passing it `wrappedProvider.provider` instead of `wrappedProvider` directly. It's because `ethers` abstracts away the low level provider calls with helper functions so it's easier for developers to use, but since not everyone uses ethers.js, Self.ID maintains a generic interface to actual provider specification, instead of the `ethers` wrapped version. We can access the actual provider instance through the `provider` property on `wrappedProvider`. `connectToSelfID` takes this Ethereum Auth Provider, and calls the

`connect` function that we got from the `useViewerConnection` hook which takes care of everything else for us.

Now, getting to the frontend a bit.

Add the following code to the end of your `Home` function

```
return (
  <div className={styles.main}>
    <div className={styles.navbar}>
      <span className={styles.title}>Ceramic Demo</span>
      {connection.status === "connected" ? (
        <span className={styles.subtitle}>Connected</span>
      ) : (
        <button
          onClick={connectToSelfID}
          className={styles.button}
          disabled={connection.status === "connecting"}
        >
          Connect
        </button>
      )}
    </div>

    <div className={styles.content}>
      <div className={styles.connection}>
        {connection.status === "connected" ? (
          <div>
            <span className={styles.subtitle}>
              Your 3ID is {connection.selfID.id}
            </span>
            <RecordSetter />
          </div>
        ) : (
          <span className={styles.subtitle}>
            Connect with your wallet to access your 3ID
          </span>
        )}
      </div>
    </div>
  </div>
);
```

For the explanation, here's what's happening. In the top right of the page, we conditionally render a button `Connect` that connects you to SelfID, or if you are already connected, it just says `Connected`. Then, in the main body of the page, if you are connected, we display your 3ID, and render a component called `RecordSetter` that we haven't yet created (we will soon). If you are not connected, we render some text asking you to connect your wallet first.

So far, we can connect to Ceramic Network through Self.ID, but what about actually storing and retrieving data on Ceramic?

We will consider the use case of building a decentralized profile on Ceramic. Thankfully, it is such a common use case that Self.ID comes with built in suport for creating and editing your profile. For the purposes of this tutorial, we will only set a Name to your 3ID and update it, but you can extend it to include all sorts of other properties like an avatar image, your social media links, a description, your age, gender, etc. For readability, we will divide this into a second React component. This is the `RecordSetter` component we used above.

Start by creating a new component in the same file. Outside your `Home` function, create a new function like this

```
function RecordSetter() {}
```

We are going to use another hook provided to us by Self.ID called `useViewerRecord` which allows storing and retrieving profile information on Ceramic Network. Let's first import it

```
import { useViewerRecord } from "@self.id/react";
```

Now, let's use this hook in the `RecordSetter` component.

```
const record = useViewerRecord("basicProfile");

const updateRecordName = async (name) => {
  await record.merge({
    name: name,
  });
```

```
};
```

We also created a helper function to update the name stored in our record (data on Ceramic). It takes a parameter `name` and updates the record.

Lastly, let's also create a state variable for `name` that you can type out to update your record

```
const [name, setName] = useState("");
```

For the frontend part, add the following at the end of your `RecordSetter` function

```
return (
  <div className={styles.content}>
    <div className={styles.mt2}>
      {record.content ? (
        <div className={styles.flexCol}>
          <span className={styles.subtitle}>Hello {record.content.name}!</span>

          <span>
            The above name was loaded from Ceramic Network. Try updating it
            below.
          </span>
        </div>
      ) : (
        <span>
          You do not have a profile record attached to your 3ID. Create a basic
          profile by setting a name below.
        </span>
      )}
    </div>

    <input
      type="text"
      placeholder="Name"
      value={name}
      onChange={(e) => setName(e.target.value)}
      className={styles.mt2}
    />
    <button onClick={() => updateRecordName(name)}>Update</button>
  </div>
);
```

This code basically renders a message `Hi ${name}` if you have set a name on your Ceramic profile record, otherwise it tells you that you do not have a profile set up yet and you can create one. You can create or update the profile by inputting text in the textbox and clicking the update button

We should be good to go at this point! Run

```
npm run dev
```

in your terminal, or refresh your webpage if you already had it running, and you should be able to connect to Ceramic Network through your Metamask wallet and set a name on your decentralized profile, and update it as well!

To look at the final code for what we have built, you can visit **Github**.

Hopefully this article could give you a sneak peek into how to use Ceramic, and help you gain an understanding of why Ceramic is important.

For verification of this level, copy your entire `did:3id:...` string from the webpage and paste it below into the verification box.

Show off your website on Discord in the showcase channel, and as always, feel free to ask any questions!

Submit Practical

Verify your smart contract address to pass the assessment for this level.

Ethereum Rinkeby | 0x... | Submit

LearnWeb3

Product

- Dashboard
- Courses
- Community
- Blog

Company

- About
- Work With Us
- We're Hiring
- Buy us a coffee

Stay up to date

Your email address