NFT english auction

Foundation Architecture

Company: Kryha
Version: 0.9
Date: 20-10-2021

1 Introduction

This document describes the foundational architecture of the NFT english auction platform. The foundational architecture describes the building blocks that the NFT english auction platform is built upon; including research results, decisions and certain important details. The goal of this document is to inform the reader of how the decentralized application is functioning and should function in the future.

Firstly, the core systems that the platform consists of are explained. Secondly, the technology stack is explained detailing the technologies and services that have been used to build the systems, followed by how they have been configured. Lastly, the application data architecture and flows are explained and made clear through diagrams.

2 Systems

2.1. Multiple canisters

In order to create a functioning auction platform on the IC, three canisters will be deployed under the same project. This approach enables flexibility and compatibility with the current IC architecture. Both backend canisters can be used independently to some degree using the candid interface, while the frontend canister requires the Auction Backend Canister to function.

2.1.1 Auction Backend Canister

This canister serves as the core of the platform by providing a database, api, and hosting business logic. Once deployed, the backend is reachable via public methods defined to perform actions on the platform as well as retrieve information. Both the Frontend canister and NFT canister interact with the auction backend via inter-canister calls. Business logic includes validating inputs and “database” operations according to the desired auction rules. Pre and post upgrade functions are also defined here in order to preserve data across canister auctions.

2.1.2 NFT Canister

The NFT Canister enables principals to create and manage NFTs to auction within the platform, it serves as a wallet that integrates with the auction canister in order to manage ownership and escrow when appropriate.

2.1.3 Frontend Canister

The Frontend Canister acts as the user interface for the platform, it allows users to see ongoing auctions, create and manage their own auctions and bids, as well as checking token balances.

3 Stack

3.1 Backend

3.1.1 Programming language

Veiling's backend is built in Motoko since it is more suited than Rust for application development on the IC. Although Rust is a more mature and utilised language in the programming community, Motoko is more suited for the environment and allows for faster development and prototyping. Furthermore, the application is not very complex from a systems point of view, so the use of Rust can't be justified on this side.

3.2 Token standard

3.2.1 NFT Standard

After having tested various NFT implementations, DepartureLabsIC was chosen as it offers the most complete and mature token implementation. Some of the criteria considered was active support and commits on Github, overall popularity, and of course, performance and ease of use. Codebase of the token can be found here. It is important to note that development of this NFT is currently in alpha, even so this currently offers the best results among the libraries that were tested.

3.2.2 Fungible Standard
TBD

3.3 Database

Given the Internet Computer's orthogonal persistence, no traditional database is needed in order to manage application data and migrations. Data is structured in stable and unstable memory in order to make the application more performant while preserving important data, this is described in detail in section 5 (Data Structure).

3.4 Frontend framework

Veiling uses the modern Single-Page Application (SPA) stack to easily compose component behaviour, styling and communications, while also providing the ease of extensibility and reusability.

3.4.1 React (^17.0.2)

The primary foundation of the stack is React. An industry standard framework, designed to build composable user interfaces. React is a free, open source and is widely supported and maintained by both Facebook and other independent developers.
One of the main features of React is the use of a Virtual DOM to ensure a performant experience, one-way data binding with regards to component hierarchy communication and a functional, declarative programming paradigm. These characteristics and related code style standards and recommendations for React and adjacent tooling are the main drivers of the architectural decisions regarding the frontend on reciChain.

3.4.2 react-router-dom(^5.3.0)

React-Router is used to take care of navigational components and functionality, as well as cross-route communication.

3.4.3 styled-components(^5.3.1)

Styled-Components is a CSS-in-Javascript library used to maintain the presentational aspects of our React components. It’s main advantage is the assurance of no scope collision between class names, the ability to use dynamic values for styles, and the segregation between presentational and behavioural parts of a component.

4. Application Configuration

4.1 Environments

The application is deployed across 4 canisters, each one with its own concern. The canisters are called: nft, fungible-token, backend, frontend. The first three are developed in Motoko, while the frontend is written in Typescript. The Motoko canisters are completely decoupled from each other, the frontend is the one responsible for making the calls and combining their behaviour. Vessel package manager is being used to handle installation and version management of external Motoko packages and Yarn is being used for JS packages.

4.1.1 NFT canister

The NFT canister is responsible for handling mint, storage and transactions of NFTs. It also keeps track of the ownership and visibility of tokens. The canister interfaces with the open source DepartureLabsIC/non-fungible-token library which is installed through Vessel package manager.

When installing this canister, a valid principal should be provided as param. That principal will be considered an admin and will be able to perform restricted calls, such as initialisation.

4.1.2 Fungible Token canister

The open source dfinance-tech/ic-token canister is being used to handle transactions and ownerships of fungible tokens used inside the application to bid for NFTs.

When installing this canister, a valid principal should be provided as param. That principal will be considered an admin and will be able to perform restricted calls. On application startup, the admin account will hold all the existing tokens.

4.1.3 Backend canister

The backend canister keeps track of all the auctions and bids and exposes methods that allow a user to perform CRUD style operations.

4.1.4 Frontend canister

The frontend canister exposes a React web application. It handles user authentication and performs calls to the previously illustrated canisters in order to handle the auction-bid flow.

5. Data architecture

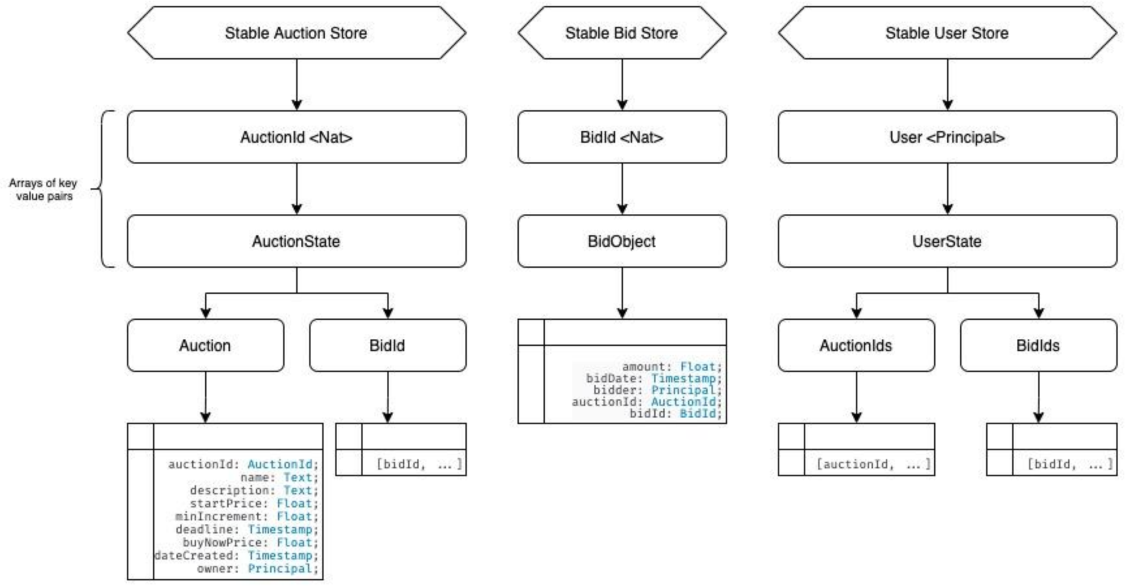
5.1 Orthogonal persistence

The Internet Computer implements orthogonal persistence, which allows for certain data variables to retain their state across canister updates. This removes the need to use an external database or implement data migration systems. In motoko, variables must be explicitly flagged as “stable” in order to persist across upgrades, any other data will be lost in the upgrade process. Find out more about Orthogonal Persistence on the IC here.

5.2 Persistent data structure

With this in mind, Veiling defines three stable variables of type array in order to persist data across upgrades. These were carefully thought out to avoid data redundancy and maximize performance.

Actions on the platform do not access these variables directly, instead [in the case of an upgrade], all data from "unstable" variables is added to it via a pre-upgrade hook. This is so more efficient data types such as HashMaps can be used to increase dapp performance. The stable data structure is as follows:



Stable Auction Store
Stable Bid Store
Stable User Store
Arrays of key value pairs
AuctionId <Nat>
BidId <Nat>
User <Principal>
AuctionState
BidObject
UserState
Auction
BidId
AuctionIds
BidIds
amount: Float; bidDate: Timestamp; bidder: Principal; auctionId: AuctionId; bidId: BidId;
auctionId: AuctionId; name: Text; description: Text; startPrice: Float; minIncrement: Float; deadline: Timestamp; buyNowPrice: Float; dateCreated: Timestamp; owner: Principal;
[bidId, …]
[auctionId, …]
[bidId, …]

5.2.1 Auctions

Holds a list of key value pairs consisting of AuctionId and AuctionState. AuctionId is of type Nat and increases by order of one in ascending order with zero being the first AuctionState. AuctionState is an object type with fields "auction" (object with auction metadata) and bidIds (array with a list of all bidIds relating to the auction object). This is meant as the main data storage for Auction data, all metadata and bidIds relating to it are saved here.

```
stable var AuctionsDb = [
 (AuctionId: Nat,
 {
   auction: {
     auctionId: AuctionId;
     name: Text;
     description: Text;
     startPrice: Float;
     minIncrement: Float;
     deadline: Timestamp;
     buyNowPrice: Float;
     dateCreated: Timestamp;
     owner: Principal;
   };
```

```
    bidIds: [BidId];
  };)
 ];
```

5.2.2 Bids

Holds a list of key value pairs consisting of BidId and AuctionState. AuctionId is of type Nat and increases by order of one in ascending order with zero being the first AuctionState. AuctionState is an object type with fields “auction” (object with auction metadata) and bidIds (array with a list of all bidIds relating to the auction object).

```
 stable var BidDb = [
  (BidId: Nat,
  {
    amount: Float;
    bidDate: Timestamp;
    bidder: Principal;
    auctionId: AuctionId;
    bidId: BidId;
  };)
 ];
```

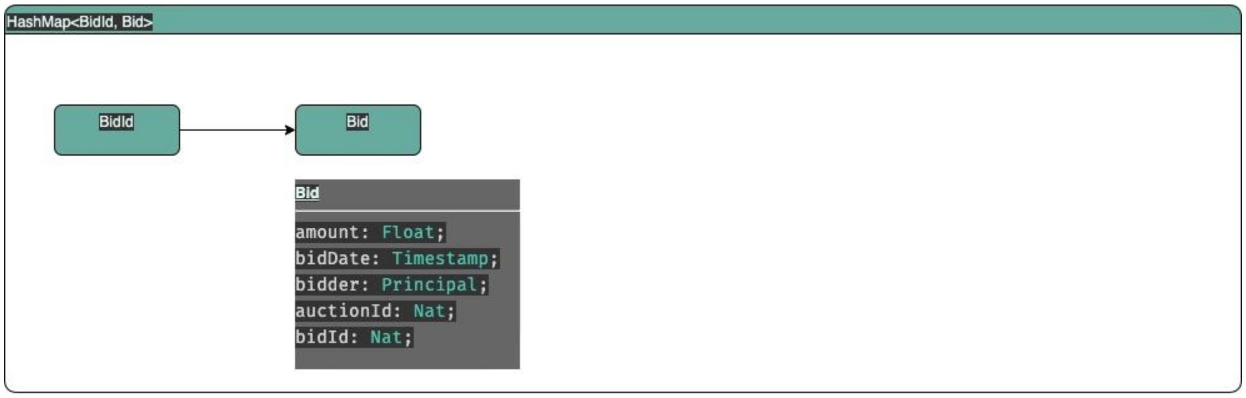
5.2.3 Users

Holds a list of key value pairs consisting of a Principal address and UserState. The principal simply identifies a user and UserState contains a list of the auction ids created by the user as well as a list of bet ids.

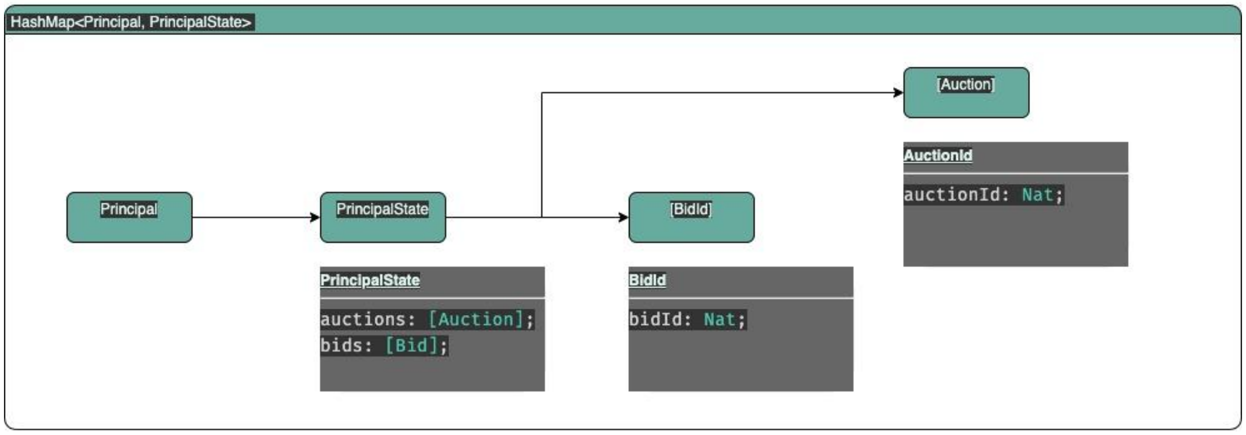
```
 stable var UserDb = [
  (User: Principal,
  {
    auctionIds: [AuctionId];
    bidIds: [BidId];
  };)
 ];
```

5.3 Performant data structure

As mentioned, in order to improve performance and perform faster lookups, a non-stable data structure is used between canister upgrades. This consists of three HashMap variables that mimic the stable variables described above. Hashmaps are data structures that can map keys to values and perform lookups using a hash function for indexing. They allow for faster lookups than arrays for Veiling’s use case. Additionally, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table, so it scales well. For all intents and purposes, the structure formed in the three hashmaps matches the key-value pairs described in the stable variable section 1:1.



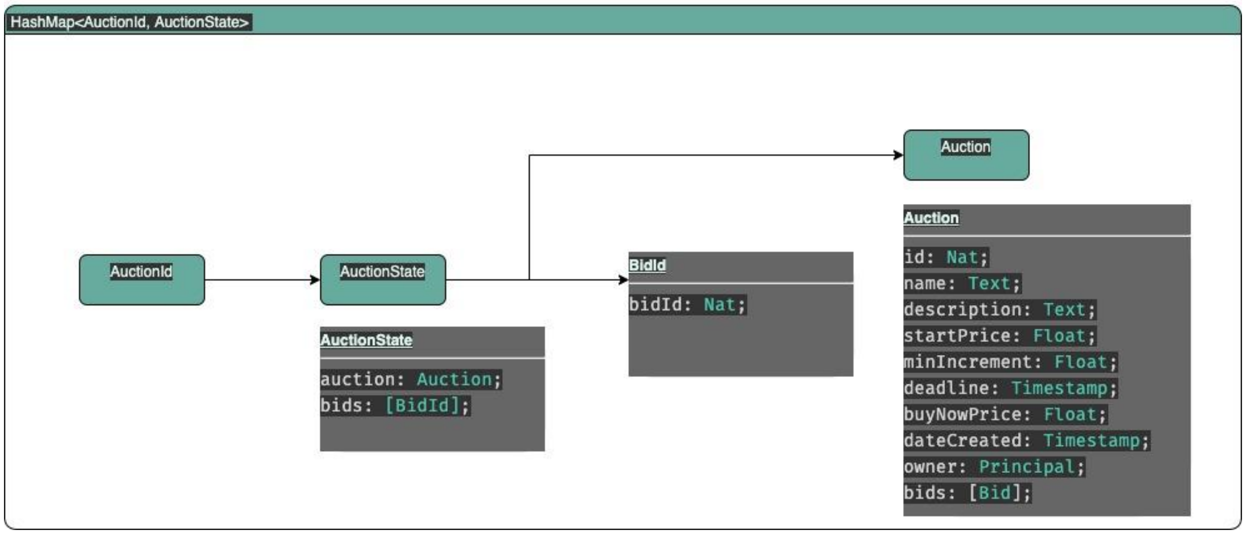
```
Bid
amount: Float;
bidDate: Timestamp;
bidder: Principal;
auctionId: Nat;
bidId: Nat;
```



```
PrincipalState
auctions: [Auction];
bids: [Bid];
```

```
BidId
bidId: Nat;
```

```
AuctionId
auctionId: Nat;
```



```
AuctionState
auction: Auction;
bids: [BidId];
```

```
BidId
bidId: Nat;
```

```
Auction
id: Nat;
name: Text;
description: Text;
startPrice: Float;
minIncrement: Float;
deadline: Timestamp;
buyNowPrice: Float;
dateCreated: Timestamp;
owner: Principal;
bids: [Bid];
```

5.4 Data flow

The following sections describe the way data flows in Veiling when performing actions such as fetching auction data, bidding data, or creating an auction.

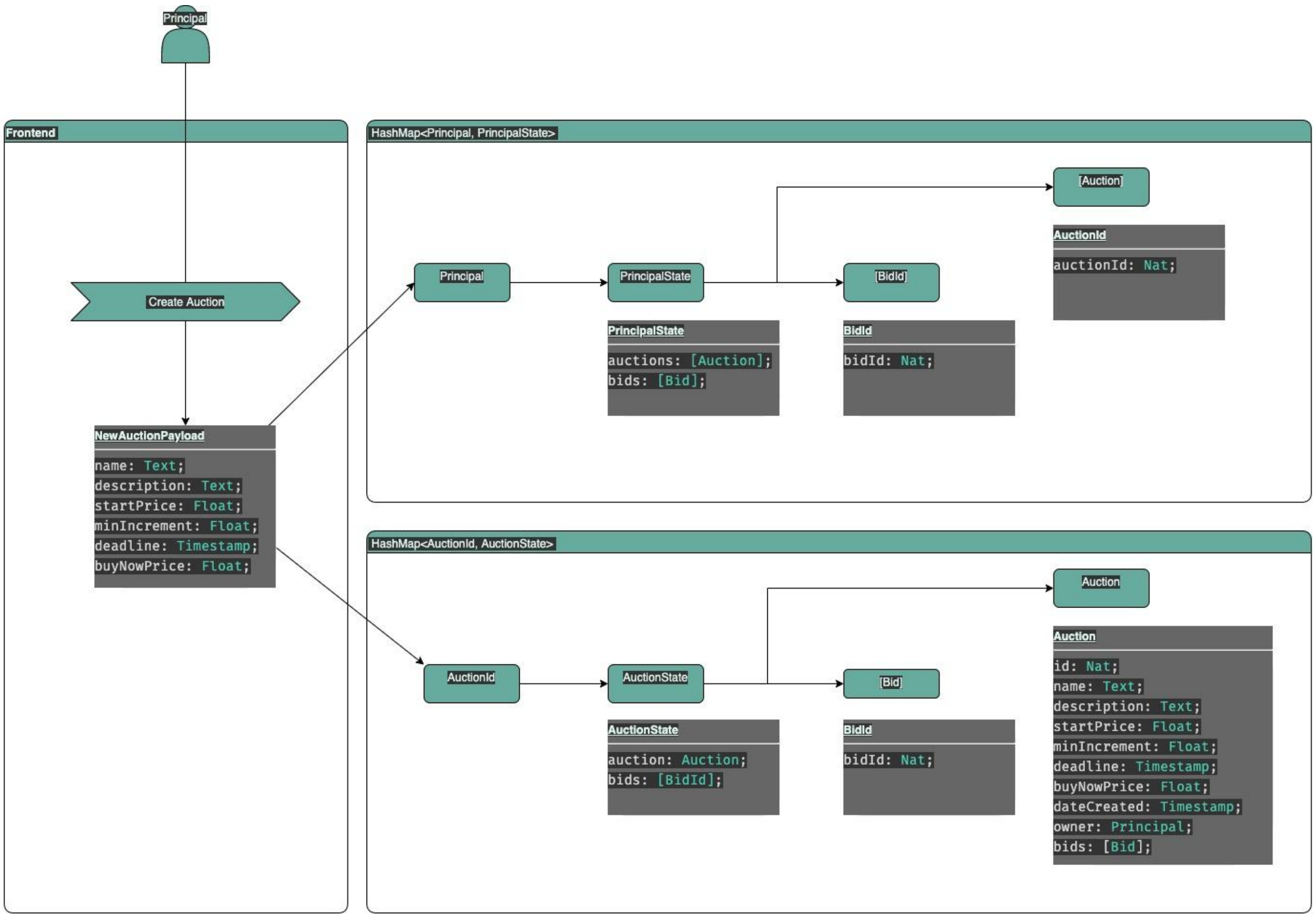
5.4.1 Create Auction

A user with a principal attached to their identity can create a new auction using the frontend interface. The auction creation is done through a React hook that calls the backend canister function newAuction:

```
 public shared({ caller }) func newAuction(auctionData: {
    name: Text;
    description: Text;
    startPrice: Float;
    minIncrement: Float;
    deadline: Timestamp;
    buyNowPrice: Float;
  }): async () { ...}
```

Type and data validation will happen both on the frontend input form and the backend function, this approach enhances usability by providing feedback and allows a principal to interact with the backend canister without relying on the frontend provided.

Once data is validated, the function will generate metadata such as the timestamp, ID, or principal address and store it in the respective data structures. The generated ID will be mapped to the auction state object consisting of auction metadata and bid id list. A reference to the auction will also be added to the user hashmap by including the auction ID under the principal’s key (a new entry is added in the case of a first time user).



5.4.2 Bid

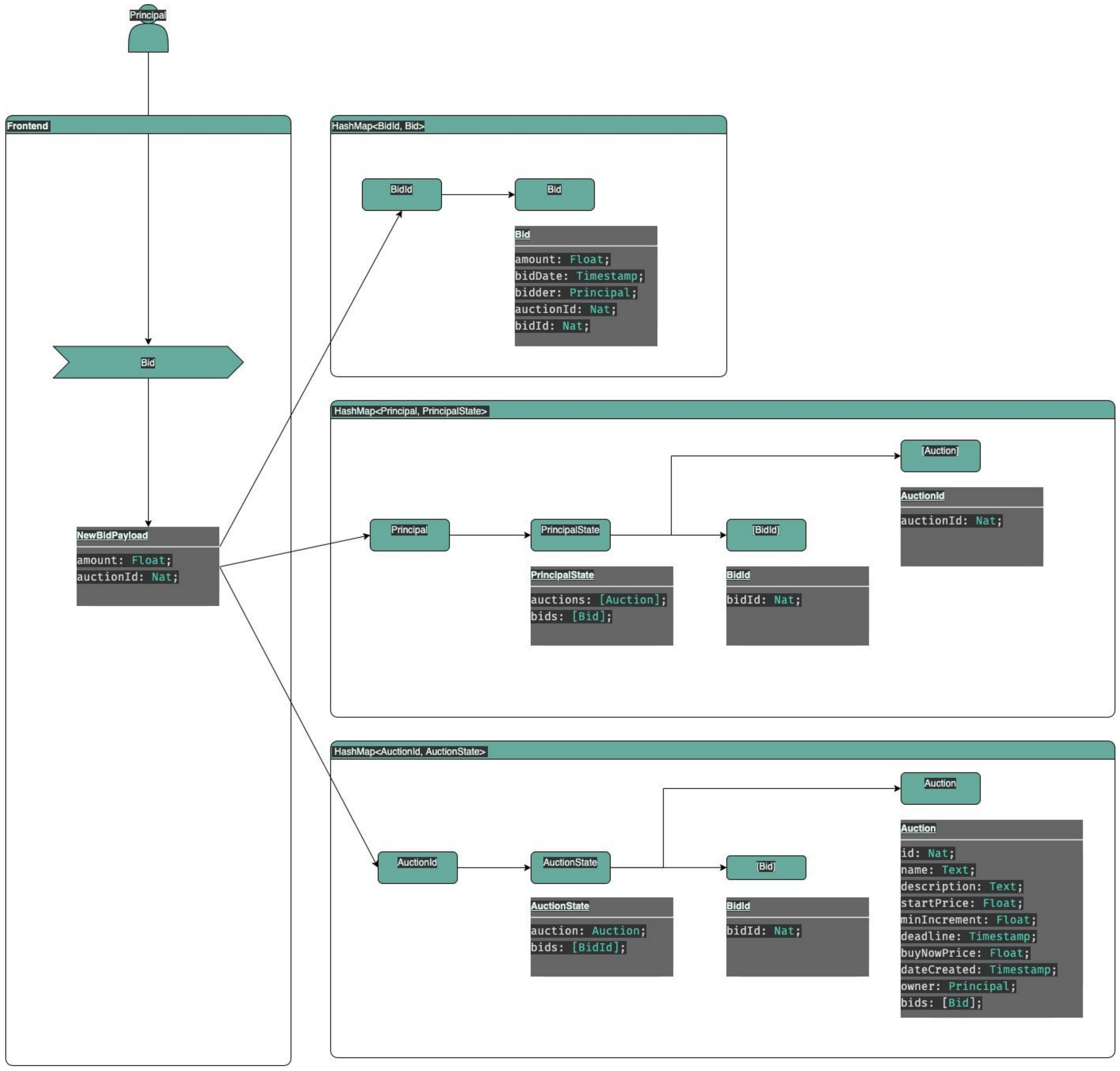
In order to place a bid in an ongoing auction, a principal can use the frontend interface to select the desired auction, input the amount, and call the backend method “bid”.

```
 public shared({ caller }) func bid(bid: {
     amount: Float;
     auctionId: AuctionId;
   };): async () { ... };
```

The bid is validated [again both in the frontend and backend] to ensure it’s in accordance with the action rules:

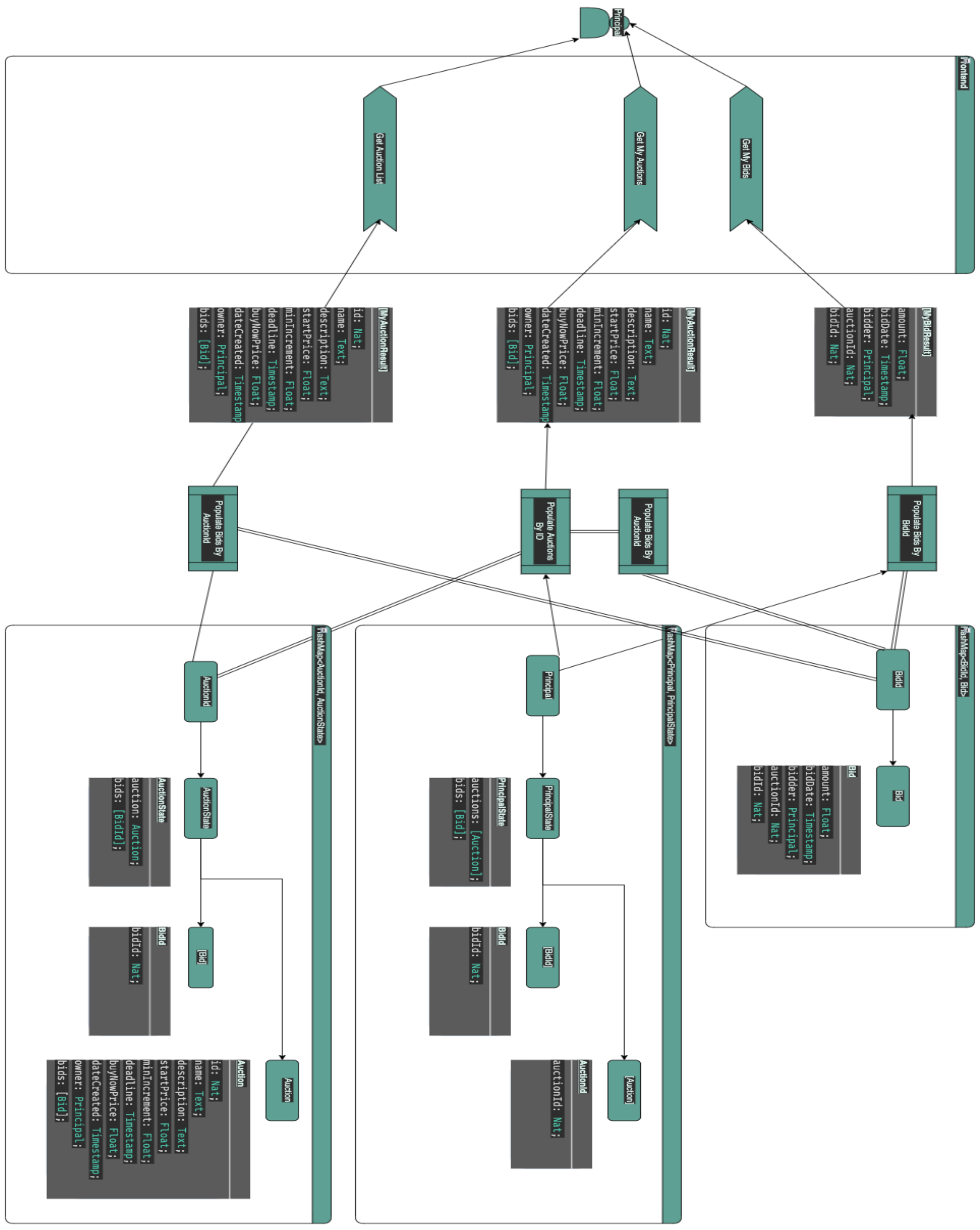
- Auction must be active, ie. “buy now” price hasn’t been reached, time limit hasn’t expired
- Principal mustn't be the owner of the auction
- Bid amount must be higher than the set starting price
- Bid amount must be smaller than “buy now” price, if set
- Bid amount must be higher than the current higher bid plus the minimum increment amount

If these requirements are met, the bid will be added to the bids hashmap, and reference by id will be set on both the auction hashmap and user hashmap accordingly.



5.4.3 Getters

The following getter functions serve auction and bid data to the React frontend or anyone wishing to interact with Veiling's canister directly:



Internal functions “populateAuctionsById” and “populateBidsById” are used to retrieve all relevant data regarding a list of Auctions or a list of bids. This helps avoid redundant storage of data and, combined with the fast lookup of hashmaps makes for a responsive system. For user specific data, the user's hashmap is queried in order to quickly find IDs relating to a user.

5.4.4 Auction completion