

PROVISIONAL APPLICATION FOR  
UNITED STATES PATENT

**BLOCKCHAIN DOMAINS**

By Inventor(s):

Matthew Everett Gould  
Reno, Nevada

Braden River Pezeshki  
Pullman, Washington

Bogdan Gusiev  
Kyiv, Ukraine

Assignee: Unstoppable Domains Inc.

VAN PELT, YI & JAMES LLP  
10050 N. Foothill Blvd., Suite 200  
Cupertino, CA 95014  
Telephone (408) 973-2585

## **DETAILED DESCRIPTION**

The invention can be implemented in numerous ways, including as a process; an apparatus; a system; a composition of matter; a computer program product embodied on a computer readable storage medium; and/or a processor, such as a processor configured to execute instructions stored on and/or provided by a memory coupled to the processor. In this specification, these implementations, or any other form that the invention may take, may be referred to as techniques. In general, the order of the steps of disclosed processes may be altered within the scope of the invention. Unless stated otherwise, a component such as a processor or a memory described as being configured to perform a task may be implemented as a general component that is temporarily configured to perform the task at a given time or a specific component that is manufactured to perform the task. As used herein, the term 'processor' refers to one or more devices, circuits, and/or processing cores configured to process data, such as computer program instructions.

A detailed description of one or more embodiments of the invention is provided below along with accompanying figures that illustrate the principles of the invention. The invention is described in connection with such embodiments, but the invention is not limited to any embodiment. The scope of the invention is limited only by the claims and the invention encompasses numerous alternatives, modifications and equivalents. Numerous specific details are set forth in the following description in order to provide a thorough understanding of the invention. These details are provided for the purpose of example and the invention may be practiced according to the claims without some or all of these specific details. For the purpose of clarity, technical material that is known in the technical fields related to the invention has not been described in detail so that the invention is not unnecessarily obscured.

Although the embodiments herein have been described in some detail for purposes of clarity of understanding, the invention is not limited to the details provided. There are many alternative ways of implementing the invention. The disclosed embodiments are illustrative and not restrictive.

In various embodiments blockchain domains replace cryptocurrency addresses with human readable names and may be powered by the Ethereum and Zilliqa blockchains. CNS and ZNS provide users full control of their domain names, which ensures permanent ownership, censorship resistance and decentralized access.

By utilizing a self-custody paradigm for domain ownership, the centralized nature of traditional domain can be removed. Once a user controls their domain, every action must user initiated. Transferring and configuring records will always be able to be done by the user without the company's interference. This is in contrast with the current security model for DNS which allows TLDs to control their subdomains at will

# Architecture Overview

This section explains the components of crypto registry, its core design principles and permission model.

This section introduces basic principles of CNS - Crypto Name Service. We don't attempt to give a deep technical explanation of them, but rather make a high-level overview, so that every reader can get an idea of how CNS works. It is assumed that a reader has basic understanding of Ethereum smart contracts and Ethereum ERC-721 token standard.

This section covers the following topics:

- Smart Contract Architecture. It gives an overview of the core smart contracts of CNS, explains how domains are managed and minted, what information about them is stored, and how users can interact with them via blockchain.
- Domain Hierarchy and Ownership describes how subdomains are managed.
- Delegating Domain Management explains the role of Transaction Processor and meta transactions in minting domains and allowing users to delegate responsibility of paying transactions costs.

Unstoppable Domains are powered by CNS. It is built on Ethereum blockchain, and at its core it's a set of smart contracts governing domains. Serving similar purpose as DNS, CNS has a lot of architectural differences that change the way how users interact with domains: while providing decentralized access, it also grants users permanent domain ownership. Unlike other services, CNS registry doesn't have the concept of renewing domains or revoking them, thus granting users full control of their second-level domains.

Every CNS domain represents an ERC-721 token. As this standard is widely supported, it makes it easier for developers to integrate with Unstoppable Domains, and users get an immediate benefit of being able to manage ownership of their domains from any wallet, exchange or marketplace that supports ERC-721.

## Smart Contract Architecture

The two central components of CNS are Registry and Resolver smart contracts. Resolver can be thought of as a *map* (or *dictionary*) of domain names to a set of records, and Registry - as a *map* of domain names to an owner address and a Resolver address.

There's only one official instance of the Registry smart contract deployed in the Ethereum Mainnet, but there are many versions of Resolver smart contracts. The relationship between Registry and Resolver can be described as one-to-many: for each domain there might be a different resolver; though in practice majority of domains is managed by the same instance (having the same address) of the latest version of Resolver smart contract. Updates to our Resolver smart contract are incremental and non-breaking, meaning that we add new features without disrupting the work of existing domains that haven't got updated.

As mentioned, every domain represents an ERC-721 non-fungible token. Every ERC-721 token is identified by a unique number, which is called token identifier. In order to make domains identifiable, we use a process called Namehashing. This process is irreversible: one can calculate a namehash of a domain having its name, but it's impossible to restore a domain name from a namehash.

Here's an example of "example.crypto" namehash:

```
0xd584c5509c6788ad9d9491be8ba8b4422d05caf62674a98fbf8a9988eeadfb7e
```

## Visualization

Figure 1 a diagram of an example to help visualize how the CNS Registry and Resolvers interact.

## Registry

Registry is one of the most essential smart contract in CNS. This is the contract that defines ownership rules, how domains are minted, provides ERC-721 token metadata and stores a list of all domains with information about them.

Registry stores the following data about each domain:

- Owner address
- Approved operator address
- Resolver address
- Domain name

As domains are identified by a namehash, we also store a domain name, apart from the other data. As it's impossible to calculate a domain name from its identifier, it allows users, that know only a namehash, lookup a real domain name.

Registry smart contract implements a set of methods allowing to mint new domains, subdomains and manage ownership.

Accounts that are allowed to mint second-level domains (like 'alice.crypto'), are called whitelisted minters. Whitelisted minters are limited to only minting new domains. They can't control domain ownership (e.g. approve or transfer a domain to another owner) and they can't control domain records. Whitelisted minters are operated by Unstoppable Domains.

A domain owner or an operator may perform the following operations with their domains:

- Transfer ownership
- Set a new resolver
- Mint a new subdomain
- Burn a domain

Users can set one trusted operator per an owned domain and one operator for their whole account, allowing managing every owned domain.

## Resolver

Resolver is a smart contract that, as its name suggests, is used for resolving domains and storing domain records. This is where domain owners store their data, such as crypto wallet addresses or IPFS records, to make a domain resolvable in the decentralized web.

The underlying data structure of Resolver can be described as a map of domain namehashes to a key-value dictionaries of records. Such structure gives users flexibility to store arbitrary records, even those that aren't specified by the Records Reference. Though in practice the data structure is a bit more complicated than that, if you're interested in implementation details, see Resolver.sol.

Resolvers allow domain owners and approved operators to edit records of their owned domains. In addition to that, Resolver provides an interface for getting standardized records, making it easier to access general information about domains.

Figure 2 a diagram illustrating example relationships between a Registry and Resolvers.

## Auxiliary smart contracts

Registry and Resolver smart contracts are not the only ones that compose Unstoppable Domains architecture. If you want to see the full list of CNS smart contracts with the links to their deployed instances and source code, see [Deployed Smart Contract Addresses](#) section. The following sections describe smart contracts that are used for optimizing resolution process and minting subdomains, which are other essential parts for understanding how CNS works.

### Proxy Reader

Proxy Reader is a smart contract that is used by our resolution libraries to actually resolve domains. Normally, it would take at least two queries to Ethereum blockchain to resolve a domain: users would call Registry, to learn a domain's Resolver address, and then they would make a call to Resolver, to read records themselves. Proxy Reader provides users a way to make just one call to get information they need.

For how resolution process works in more details, see [Resolving Domain Records](#) section.

### Minting Controller

Minting Controller is a smart contract that is allowed to mint second-level domains. The deployed version of Registry smart contract allows only Minting Controller to mint domains. This smart contract is used by Whitelisted Minter as a proxy, which is the actual smart contract that gets called by minter accounts.

The mentioned contracts are operated by Unstoppable Domains and don't have any methods that can be executed by a third party.

## Domain Hierarchy and Ownership

There are plenty ways of setting up ownership model in CNS. A domain can be owned by either an external address, that is accessed with a private key, or an internal one, i.e. smart contract. Managing domains with smart contracts opens up a lot of possibilities in how to organize ownership. Domain management can be set up to a multisig wallet or it can be equally shared among a group of administrators – these are just two simple examples, anything else expressible with Ethereum smart contracts is possible as well.

Subdomains of any level are subjects to the same set of rules as second-level domains in CNS. Subdomains can be transferred or burned by both a direct owner and a domain zone owner (i.e. parent domain owner). Only a direct domain owner can set Resolver address and manage domain records.

Such ownership model might not be suitable for every user. One may want to mint subdomains that are irrevocable and can be transferred only by a direct owner – this and other models are possible if a domain zone is owned by a smart contract. For more details, see Alternative Ownership Models below.

**Note** that owner of 'crypto' top-level domain is set to a burn address, that means that only direct owners can transfer or burn their second-level domains. In order to mint second-level domains we use a different mechanism, which doesn't rely on domain ownership: for more information, see Minting Controller section.

## Alternative Ownership Models

Solidity smart contracts allow other models of managing subdomains to be employed. One of the ways is to use DomainZoneController smart contract, which is supported by Unstoppable Domains team.

DomainZoneController suggests the model of ownership where subdomains can be managed only by their owners. Minting subdomains can be performed using whitelisted accounts, that can be added or renounced by smart contract admins.

Apart from minting subdomains, DomainZoneController provides functionality to set a Resolver address and manage records of a domain zone itself – these actions can be performed only by whitelisted accounts as well.

Internal addresses can't be accessed directly by a wallet, which implies that domains owned by smart contracts can only be managed with rules defined by smart contracts themselves. For example, one cannot transfer a subdomain if such functionality isn't defined by smart contract set of methods.

## Delegating Domain Management

CNS allows users to delegate executing certain transactions to accounts that aren't domain owners. Registry and Resolver smart contracts implement methods that make use of Meta Transactions. One of the use-cases for using Meta Transactions is delegating blockchain calls, that use gas, to other accounts. For domain owners it

means that they don't need to have funds on their accounts to manage their domains, if another party takes responsibility to pay for executing transactions.

Unstoppable Domains uses a system that we call Transaction Processor. Transaction Processor makes it possible for users to mint and manage their domains without having to care about their wallet's balance. Under the hood it represents a queue-based job processor, that sends transactions from accounts owned by Unstoppable Domains.

In general, there are two types of transactions that Transaction Processor takes care of:

- Minting domains
- Managing domains (transferring, setting Resolver address, modifying records)

**Minting domains** is generally triggered when a user claims a domain via Unstoppable Domains website. This action doesn't require a domain owner's signature, since minting second-level domains is controlled by Unstoppable Domains.

**Managing domains**, on the contrary, can be performed only with a domain owner's permission. Each delegated transaction, that modifies either owner address, Resolver address, or domain records, requires a domain owner's signature.

Delegating CNS transactions is not necessarily coupled with Transaction Processor. It's possible to execute any write operation that supports Meta Transactions from any Ethereum account, given that a domain owner provides a signature for such call.

For technical details of delegating transactions in CNS, see Meta Transactions section.

## Namehashing

Namehashing is an algorithm that converts a domain name in a classical format (like `www.example.crypto`) to ERC721 token id. All .crypto ecosystem contracts accept domain name as a method argument in a form of ERC721 token. Namehashing (e.g., Ethereum EIP-137 standard). See the standard for a text description of the algorithm.

One can verify his implementation of the namhash algorithm using the following reference table:

| Domain Name            | ERC721 Token                                                                   |
|------------------------|--------------------------------------------------------------------------------|
| .                      | 0x0000000000000000000000000000000000000000000000000000000000000000<br>00000000 |
| crypto                 | 0x0f4a10a4f46c288cea365fcf45cccf0e9d901b945b9829ccdb54c10d<br>c3cb7a6f         |
| example.crypto         | 0xd584c5509c6788ad9d9491be8ba8b4422d05caf62674a98fbf8a9988<br>eeadb7e          |
| www.example.cryp<br>to | 0x3ae54ac25ccd63401d817b6d79a4a56ae7f79a332fe77a98fa0c9d10<br>adf9b2a1         |
| a.b.c.crypto           | 0x353ea3e0449067382e0ea7934767470170dcfa9c49b1be0fe708adc4<br>b1f9cf13         |

## Reverse Lookup

Fundamentally namehashing is built to be a one way operation. It recursively hashes the labels using the SHA-256 hash function.

If one possesses a precomputed table of all hashes and corresponding domains reverse lookups are possible. This table can be reconstructed using the events on the CNS Registry `NewURI` event.

## JSON RPC

Crypto registry remembers all the domain names that were ever minted with their corresponding namehash: see Registry.sol. That makes it possible to obtain an original domain name from a namehash via ETH RPC call to tokenURI function of Registry.sol.

## Public API

There is a way to obtain domain information by namehash from APIs. In order to do this you need to call `https://unstoppabledomains.com/api/v1/{namehash}`  
Example:

```
GET request:
{
  "addresses": {},

```

```

    "whois": {},
    "ipfs": {},
    "gundb": {},
    "social": {},
    "meta": {
      "domain": "crypto",
      "namehash":
"0x0f4a10a4f46c288cea365fcf45cccf0e9d901b945b9829ccdb54c10dc3cb7a6f",
      "owner": "0x00000000000000000000000000000000dead",
      "type": "CNS",
      "ttl": 0
    },
    "records": {}
  }
}

```

Response:

```

{
  "addresses": {},
  "whois": {},
  "ipfs": {},
  "gundb": {},
  "social": {},
  "meta": {
    "owner": "0x00000000000000000000000000000000dead",
    "type": "CNS",
    "ttl": 0,
    "domain": "crypto",
    "namehash":
"0x0f4a10a4f46c288cea365fcf45cccf0e9d901b945b9829ccdb54c10dc3cb7a6f"
  },
  "records": {}
}

```

## Resolving Domain Records

Resolving a domain is a process of retrieving a domain records when the domain name and required record names are given. There is no limitation on who can read domain records on Registry side. Anyone having access to Ethereum Node on the mainnet can resolve a domain.

This section describes resolving domain records with making calls to Ethereum smart contracts, using Ethereum JSON RPC. For developers, who are interested in more high level solution, it might be more convenient to use resolution libraries instead.

Resolving a domain requires software to have access to Ethereum network. For more information, see [Configuring Ethereum Network connection](#) below.

The simplest way to resolve a domain with Ethereum JSON RPC is to make a readonly call to ProxyReader smart contract. ProxyReader provides an API that allows users to resolve domains making just one call, passing only keys of records and a domain namehash. Without ProxyReader it would require executing at least two calls: one to obtain a domain resolver address and another one to get the records themselves. With ProxyReader it all happens under the hood.

An example in JavaScript of getting two records (using Ethers Project library):

```
const proxyReaderAddress = '0x7ea9Ee21077F84339eDa9C80048ec6db678642B1';
// Partial ABI, just for the getMany function.
const proxyReaderAbi = [
  'function getMany(string[] calldata keys, uint256 tokenId) external view
  returns (string[] memory)'
];
const proxyReaderContract = new ethers.Contract(
  proxyReaderAddress,
  proxyReaderAbi,
  provider,
);

const domain = 'brad.crypto';
const tokenId = namehash(domain);
const keys = ['crypto.ETH.address', 'crypto.BTC.address'];

const values = await proxyReaderContract.getMany(keys, tokenId);
console.log(values);
// [
//   '0x8aaD44321A86b170879d7A244c1e8d360c99DdA8',
//   'bclq359khn0phg58xgezyqsuuaha28zkwx047c0c3y'
// ]
```

Reference:

- `namehash` - namehashing algorithm implementation. See Namehashing section.

Figure 3 is a diagram illustrating an example of communications between components for providing domain records via a proxy reader smart contract.

See Records Reference section for more information about the standardized records.

## Record Value Validation

Crypto resolver doesn't have any built-in record value validation when it is updated for two reasons:

- Any validation would require additional gas to be paid
- Solidity is special purpose programming language that doesn't have any built-in data validation tools like Regular Expressions

Any domain management application must perform record format validation before submitting a transaction. However, there is no guarantee that all management application will do it correctly. That is why records must be validated when domain is resolved too.

See Records Reference section for more information for the validator of each record.

## Configuring Ethereum Network connection

Domain Resolution Configuration at low level has 3 configuration parameters:

1. Ethereum JSON RPC provider
2. Ethereum CHAIN ID
3. Crypto Registry Contract Address

Ethereum JSON RPC provider is an API implementing Ethereum JSON RPC standard. Usually, it is given in a form of HTTP API end point. However, other forms may exist in case when ethereum node is launched locally. Unstoppable domains recommends Cloudflare Ethereum Gateway an ethereum node service provider.

Ethereum CHAIN ID is an ID of ethereum network a node is connected to. Each RPC provider can only be connected to one network. There is only one production network with CHAIN ID equal to `1` and called `mainnet`. Other networks are only used for testing purposes of different kinds. See Ethereum EIP-155 for more information. CHAIN ID of an ethereum node can be determined by calling `net version` method on JSON RPC which should be used as a default when only JSON RPC provider is given. Crypto Registry Contract Address is an actual address of a contract deployed. There is only one production registry address on the mainnet: `0xD1E5b0FF1287aA9f9A268759062E4Ab08b9Dacbe`. This address is used as a default for production configuration.

# Records Reference

Records Reference is an overview of all standardized resolver record that have a defined interpretation. Any other custom records are not forbidden to be set, however their interpretation remains custom to specific clients. See Resolving Domain Records section and Resolve in the Architecture Overview section for more information on domain records.

## List of Records

| Domain Name                                | Description                                                                          | Format                | Example                                        |
|--------------------------------------------|--------------------------------------------------------------------------------------|-----------------------|------------------------------------------------|
| <code>crypto.ETH.address</code>            | Ethereum Address to receive crypto payments                                          | 0x[0-9a-fA-F]{40}     | 0x0f4a10a4f46c288cea365fcf45cccf0e9d901b94     |
| <code>crypto.BTC.address</code>            | Bitcoin Address to receive crypto payments                                           | [0-9a-zA-Z]{32}       | 1Nb7Mt1EqUqxrrAdmefUovS7aTgMUf2A6m             |
| <code>crypto.&lt;TICKER&gt;.address</code> | Crypto currency address of the ticker.                                               |                       |                                                |
| <code>browser.preferred_protocols</code>   | Protocols that browser should prioritize to display content for                      | JSON serialized array | ["ipfs","http"]                                |
| <code>browser.redirect_url</code>          | An URL where a browser should redirect a person if no other resolution method found. | RFC-1738              | http://example.com/home.html                   |
| <code>dweb.ipfs.hash</code>                | IPFS network content hash                                                            | [0-9a-zA-Z]{46}       | QmVaAtQbi3EtsfpKoLzALm6vXphdi2KjMgxEDKeGg6wHvK |
| <code>dweb.bzz.hash</code>                 | Swarm network content hash                                                           |                       |                                                |
| <code>dweb.&lt;PROTOCOL&gt;.hash</code>    | Any other distributed content network content hash                                   |                       |                                                |
| <code>dns.ttl</code>                       | Default TTL setting for all DNS records                                              | \d+                   | 128                                            |

|                                         |                                                            |                                 |                                                             |
|-----------------------------------------|------------------------------------------------------------|---------------------------------|-------------------------------------------------------------|
| <code>dns.A</code>                      | DNS A record IP addresses                                  | JSON<br>serializ<br>ed<br>array | <code>["10.0.0.1","10.0.0.2"]</code>                        |
| <code>dns.A.ttl</code>                  | TTL setting for all A records                              | <code>\d+</code>                | 128                                                         |
| <code>dns.CNAME</code>                  | DNS CNAME record IP addresses                              | JSON<br>serializ<br>ed<br>array | <code>["example.com."]</code>                               |
| <code>dns.CNAME.ttl</code>              | TTL setting for all CNAME records                          | <code>\d+</code>                | 128                                                         |
| <code>dns.&lt;RECORD&gt;</code>         | Specified DNS record values.                               | JSON<br>serializ<br>ed<br>array |                                                             |
| <code>dns.&lt;RECORD&gt;.ttl</code>     | TTL setting for corresponding type of records              | <code>\d+</code>                | 164                                                         |
| <code>ipfs.html.value</code>            | Deprecated: use <code>dweb.ipfs.hash</code> instead.       | <code>[0-9a-zA-Z]{46}</code>    | <code>QmVaAtQbi3EtsfpKoLzALm6vXphdi2KjMgxEDKeGg6wHvK</code> |
| <code>ipfs.redirect_domain.value</code> | Deprecated: use <code>browser.redirect_url</code> instead. | RFC-1738                        | <code>http://example.com/home.html</code>                   |

## Managing Domain Records

Domain records can be managed via default public resolver. One can develop its own custom resolver with any management permissions defined.

Figure 4 is a diagram illustrating an example of domain record management.

## Resolver Record Types

Records on top level are stored in a simple key-value pair mapping string to string. CNS doesn't forbid a user to assign any record to any value. However, there is a list of standard records that have a defined standard interpretation by clients. A full list of standardized records can be found in the Records Reference section.

Standard record keys are split by namespaces with `.` used as a separator. Main namespaces are:

- `crypto.*` - records related to crypto payments
- `dns.*` - DNS records
- `dweb.*` - records related to distributed content network protocols
- `browser.*` - hint records for web browsers

## Crypto Payments Records

One of essential feature of crypto domains is the ability to make specify a domain instead of a destination address for your crypto payment. Cryptocurrency wallets that need this feature should revolve a domain to crypto address under the hook in the same way browser resolves a domain to IP address.

All crypto addresses are stored within `crypto.*` namespace. Each currency address is stored as `crypto.<TICKER>.address` record. Example: Bitcoin address is stored in `crypto.BTC.address`. Addresses are stored in plain text format according to an address space standard established by each individual currency namespace can contain additional currency specific attributes to facilitate payment delivery. Example: Ripple Destination Tag of the XRP ledger. However, key names for those attributes are not yet standardized.

Some tickers of very popular crypto currencies are not yet standardized. Example: [LINK](#) for Chainlink Crypto. Standardized list of tickers can be found in [SLIP-0044](#) of SatoshiLabs. However, more extended list of conventional tickers is available at <https://github.com/crypti/cryptocurrencies/blob/master/cryptocurrencies.json>.

Example crypto records setup:

| Key                             | Value                                                    |
|---------------------------------|----------------------------------------------------------|
| <code>crypto.ETH.address</code> | <code>0xD1E5b0FFf1287aA9f9A268759062E4Ab08b9Dacbe</code> |
| <code>crypto.BTC.address</code> | <code>bc1qkd4um2nn2uyzmsch5y86wsa2pfh8xl445lg9nv</code>  |
| <code>crypto.ZIL.address</code> | <code>zill1yu5u4hegy9v3xgluweg4en54zm8f8auwxu0xxc</code> |

## Resolver Administrative Patterns

### Default Resolvers' Record Ownership Style

The default Unstoppable resolver allows users to manage all domain records for any address given a permission over domain utilizing the ERC721 "Transfer Mechanism". This enables a subset of addresses to manage the domain on your behalf. By default we give the permission to do this to every address that can already transfer ownership of the domain. These include:

- Owner address of a domain
- Approved address for a domain
- Owner's operator addresses

This enables Users to enable smart contract applications and other users to manage one or all of the domains, while still retaining primary ownership of their domain. These smart contracts can be programed in such a way that they only change specified records. An Oracle Integration works in this way. Here is an example Flow:

1. Users grant operator access to all of their domains to the Oracle Contract.
2. Oracle detects an event off chain.
3. The Oracle sets a record inside the resolver contract.

See ERC721 on how those permissions can be granted and revoked. Any records change is submitted as a Ethereum Blockchain Transaction. And how record management can be done via Resolver methods (e.g., see IResolver.sol).

What is important about this model is you can delegate trust to change records to smart contracts. This enables users to interact with applications that could store keys and other information on a domain. Making the domains a metadata repository or cross application identifier.

## Alternative Ownership Styles

Resolvers can be made with custom logic. For example users with a large amount of domains might want to deploy a custom `Owned` resolver contract. See Ethereum EIP-173. Where there is only one set of records, and only the owner can set them.

```
pragma solidity ^0.7.0;

import "./Owned.sol";

contract CustomResolver is Owned {
    mapping (string => string) internal _records;

    function set(string memory key, string memory value) public onlyOwner {
        _records[key] = value;
    }

    function get(string memory key, uint256 /* unused tokenId */) public
    returns (string memory) {
        return _records[key];
    }

    // ...
    // Rest of the IResolver.sol methods
    // ...
}
```

This smart contract will read from the same set of records every time, allowing for efficient updating of multiple domains at once. See more in Deploying Custom Resolvers section.

## Presets

The resolver also implements a preset mechanism. The records inside the Resolver are stored as a nested mapping.

```
Token ID -> Preset ID -> Key -> Value
```

This nested structure allows users to configure domains on the un-enumerable Ethereum mappings. Typically it is expensive and unreliable to store an enumerable data structure on Ethereum. To get around this domains store a preset that corresponds to a record set. This means that users can change the preset on the domain to get an entirely different set of records.

Currently `reset` and `reconfigure` are the only methods exposed on this version of resolver.

- The `reset` method clears the domain's records by setting the preset on the domain to the timestamp when the transaction was mined e.g. `blockchain.timestamp`.
- The `reconfigure` method first `reset`'s the domain then configures a new set of records.

## Pre-configuring Records

The Default Resolver allows the Unstoppable Minting EOAs to mint and preconfigure domains in one step. This `preconfigure` method only lets the Minting EOAs configure unowned domains not names already minted to the CNS Registry.

## Meta Transactions

Most Registry and Resolver methods have meta-transaction support. Generally, meta-transactions allow users to sign messages to control their domains that are then submitted to the registry by a different party. This enables Unstoppable to submit transactions on behalf of users, so that the user can still manage their domains in a self-custodial capacity without any gas.

Meta-transactions work by having users sign function calls along with a nonce. Sending that signed function call over to a different party. That party calls the meta-transaction enabled function on the Registry or Resolver. For most management methods, there is a method with meta-transaction support that has `For` suffix at the end. The meta-transaction method then checks the permission for a domain against the address recovered from the signed message sent to the function, unlike base method that checks it against the submitter of the transaction e.g. `msg.sender`.

Figure 5 is a diagram illustrating an example meta transaction flow.

Concretely `resetFor` is a meta-transaction version of `reset`. This method has an additional **signature** argument as a last parameter beside all original parameters. Note that the meta-transaction versions of Registry functions are implemented in controller contract, not in the registry itself.

A source code of signature validation can be found in `SignatureUtil.sol`

## Token Nonce

Meta transaction methods are bound to names via their nonce (instead of Account Nonce of traditional transactions). It protects from Double-spending in the same way as account based nonce in traditional transactions.

Figure 6 is diagram illustrating an example of how replay attacks can be used to exploit domains.

A nonce is simply a transaction counter for each token. This prevents replay attacks where a transfer of a token from A to B can be replayed by B over and over to continually revert the state of the name back to a previous state. This counter must increment up by 1 each time a state transition happens to a token. Token based nonces can be used to prevent mis-ordering of transactions in a more general sense as well. This prevents front running non fungible assets and this property can also be used to batch transactions securely as well.

Figure 7 is diagram illustrating an example of using nonces to prevent mis-ordering of transactions.

## Meta transaction signature generation

Meta transaction requires two signatures: one passed as a method argument and one classical. Classical signature is generated in a standard way. Meta signature requires a domain owner (or a person approved by owner) to sign a special message formed from:

- Domain based meta-transaction nonce
- Function Selector (of Solidity, an object-oriented, high-level language for implementing smart contracts) of the original method
- Original Method parameters (the one without signature)

Example Signature generation for a `reset` method call for a domain:

```
const domain = 'example.crypto';
const methodName = 'reset';
const methodParams = ['uint256'];
const contractAddress = '0xb66DcE2DA6afAAa98F2013446dBCB0f4B0ab2842';
// Can be different or the same as contractAddress
const controllerContractAddress =
'0xb66DcE2DA6afAAa98F2013446dBCB0f4B0ab2842';
const tokenId = namehash(domain);

function generateMessageToSign(
  contractAddress: string,
```

```

signatureContract: string,
methodName: string,
methodParams: string[],
tokenId: string,
params: any[],
) {
  return solidityKeccak256(
    ['bytes32', 'address', 'uint256'],
    [
      solidityKeccak256(
        ['bytes'],
        [encodeContractInterface(contractAddress, method, methodParams,
params)],
      ),
      controllerContractAddress,
      ethCallRpc(controllerContractAddress, 'nonceOf', tokenId),
    ],
  );
}

const message = generateMessageToSign(
  contractAddress,
  signatureContractAddress,
  methodName,
  methodParams,
  tokenId,
  [tokenId]
);

```

#### Functions Reference:

- `namehash` - Namehashing Function algorithm implementation (see Namehashing section)
- `ethCallRpc` - Ethereum `eth_call` JSON RPC implementation
- `encodeContractInterface` - Solidity ABI interface parameters encoder
- `solidityKeccak256` - Solidity ABI parameters encoder

## Deploying Custom Resolver

Custom Resolver is a way to implement a flexible permission model over domain records. This may include:

- Community owned domains - individual members of a community decide on a domain website content
- Domain leasing - a domain is temporary managed by a different authority and then comes back to original authority

Custom resolver can be implemented as a separated contract and its address can be assigned as a domain resolver via `Registry#resolveTo`. Custom resolver implements interfaces `IResolverReader` (see `IResolverReader.sol`) and Ethereum ERC165

## Resolving Domains in a Browser

This section describes the recommended way to resolve blockchain domains within a classical HTTP Web Browser or a Dapp Browser.

### End user features

Here are some of the end user scenarios that should give an idea which features should be available in a browser supporting crypto domains.

### HTTP Website Browsing

1. Given a blockchain domain has a DNS record configured.
2. When user enters the domain name into a browser address bar.
3. Then the browser resolves the domain and gets the specified DNS records.
4. Then the browser requests and displays the content using DNS protocol and HTTP protocol.

### Distributed Website Browsing

1. Given a blockchain domain has an dweb content identifier record configured (e.g. IPFS hash).
2. When user enters the domain name into a browser address bar.
3. Then the browser resolves the domain and gets the content hash of a domain.
4. Then the browser retrieves the content by the hash using a related protocol and displays the content.

Figure 8 is a diagram illustrating an example overview of reading DWeb website from Ethereum and decentralized network.

## Domain Level Redirect

1. Given a blockchain domain has a redirect url and IPFS hash configured, and a user's browser doesn't support IPFS protocol.
2. When the user enters the domain name into a browser address bar.
3. Then the browser resolves the domain and gets redirect url and IPFS hash records.
4. Then the browser redirects a user to the redirect URL because IPFS protocol is not supported.

## Resolution Configuration

1. Given a user that want to change its ETH provider service.
2. When user goes to browser settings crypto domains section.
  - o User changes ethereum node URL from default to any other.
  - o User changes Registry Address for each support crypto registry.
  - o User changes network for ethereum node.
  - o User changes DNS gateway
  - o User changes Dweb gateway
3. Then the browser uses new settings to make requests to ethereum blockchain
  - o If network is not specified explicitly, it can be retrieved from the ethereum node URL.
  - o If Registry Address is not specified, it can use a default for specified network

Figure 9 is a diagram illustrating an example of configuring a DNS gateway.

## Content Display Protocol

In addition to base browser content display protocol like `http` blockchain domains can also be configured for distributed content protocol like `ipfs`. Here is the list of content display protocols that can be associated with a crypto domain:

- Traditional
  - o HTTP
  - o HTTPS
  - o FTP
- Distributed
  - o IPFS (InterPlanetary File System) defining `ipfs://` protocol
  - o Swarm (a distributed storage platform and content distribution service) defining `bzz://` protocol

A browser may support any subset of traditional or distributed protocols that would still make crypto domains websites displayable.

## Gateway to simplify the integration

While it is possible to make resolve a domain via a call to ETH RPC and support distributed content protocols in a browser, it might be easier to make those call via gateways using protocols already supported by all browsers: HTTP and DNS. A gateway may simplify the integration to a browser, but comes at the downside of decreased decentralization (if gateway is hosted by third party) or complex user experience if the gateway is hosted by the user.

There are 2 possible gateways for each of the problem:

- Distributed content (Dweb) gateway
- Resolution over DNS gateway

Figure 10 is a diagram illustrating an example of overview of utilizing a DNS gateway and a Dweb gateway.

### Distributed content gateway

A gateway is an HTTP Server that acts as a proxy between HTTP and distributed content protocol. Basic functionality of such a gateway:

1. Receive HTTP request to crypto domain (like `http://example.crypto`)
2. Resolve a domain into crypto records
3. Get the content based on Browser Resolution Algorithm (see Browser Resolution Algorithm section)
4. Return the content to client via HTTP

### Resolution over DNS gateway

A gateway is a DNS Server that resolves not just traditional domains but also `.crypto` domains. Basic functionality of such a gateway:

1. Receive a domain resolution request
2. Resolve a domain using classical DNS system if is in classical TLD (like `.com`)

3. Resolve a domain using Browser Resolution Algorithm if it is in crypto TLD (see Browser Resolution Algorithm section)
  - If a domain is set using DNS, transform crypto DNS records into classical records
  - If a domain is set using distributed content (see Browser Resolution Algorithm section)
    - If client requests `A` record, resolve to Distributed Content Gateway IP Address
    - If client requests `txt` record, resolve to all crypto records in JSON encoded key-value format
4. Send resolution to client

Figure 11 is a diagram illustrating an example of resolving a DWeb website via a DNS gateway and a DWeb gateway.

## Browser Resolution Algorithm

This section explains how different domain record configurations should be interpreted by browsers.

Figure 12 is a diagram illustrating an example of resolving a DWeb website by direct reading from Ethereum and Decentralized network.

A browser can select a protocol it has a support for. If a domain is configured for multiple protocols, it should prioritize a protocol based on `browser.preferred_protocols` record that can be set to a list of the defined protocols.

If `browser.preferred_protocols` is not set, a browser should use the following value as a default `["bzz", "ipfs", "https", "http", "ftp"]`. If `browser.preferred_protocols` is set but is not complete a browser should fulfill the absent protocols at the end in the default order specified above. A domain can have a single content identifier for each distributed protocol stored in `dweb.<protocol>.hash`. EX: `dweb.bzz.hash` for Swarm's `bzz` protocol. See Dweb Records below for more information.

If none of `dweb` hash records is set, a browser should fall back to DNS resolution that is set within `dns.*` namespace.

Generally browsers automatically add `http://` prefix for any domain in the address bar if the protocol is not specified explicitly by a user. In case of blockchain domain names (assuming a browser supports many protocols), it is preferred to determine a protocol only after resolving domain records.

Figure 13 is a diagram illustrating an example flowchart for browser resolution.

## Records related to browser resolution

All records related to browser resolution are stored within these namespaces:

- `dns.*` - for traditional DNS records
  - `dweb.*` - for distributed content records
  - `browser.*` - hint records to help browser determine a preferred content display method
- If you are looking for a way to get records associated to a domain, see Resolving Domain Records section.

### DNS records

Resolver records may contain classical DNS records besides other records. In order to distinguish those from other crypto records, the `dns.*` namespace is used. So DNS `A` corresponds to `dns.A` crypto record. Any listed DNS record described in RFC standards is supported. All record names must follow upper case naming convention.

As crypto resolver doesn't support multiple records with the same key, but DNS does allow that. Therefore, DNS record value must always be stored as JSON serialized array of strings. Example 1: a domain that needs one `CNAME` record set to `example.com`, must be configured as one crypto record `dns.CNAME` set to `["example.com"]`. Example 2: a domain that needs two `A` records set to `10.0.0.1` and `10.0.0.2` must be configured as one crypto record `dns.A` set to `["10.0.0.1", "10.0.0.2"]`.

No other data transformation is required when converting a traditional DNS record into Crypto record other than aggregating records with the same name to one record using serialization as JSON array of strings.

Crypto records do not have a domain name associated to them. That is why there is no feature of storing your subdomain records inside a parent domain. Example: `www.example.com` record can only be set inside a resolver of `www.example.com` but never inside `example.com`.

A recommended way to display content in a browser for crypto domains is explained in Resolving Domains in a Browser section.

## TTL

Records TTL can be set for all records or for individual type of record. TTL for all records can be set in `dns.ttl`. TTL for individual record type can be set in `dns.<RECORD>.ttl`. If `ttl` for individual record type is not set, a default `dns.ttl` need to be applied.

Example crypto records setup:

| Record    | Value                        |
|-----------|------------------------------|
| dns.A     | ["10.0.0.1", "10.0.0.2"]     |
| dns.A.ttl | 168                          |
| dns.AAAA  | ["2a00:1450:401b:805::200e"] |
| dns.MX    | ["10 aspmx.example.com."]    |
| dns.ttl   | 128                          |

Would be transformed into the following DNS records:

| Record | Value                    | TTL |
|--------|--------------------------|-----|
| A      | 10.0.0.1                 | 168 |
| A      | 10.0.0.2                 | 168 |
| AAAA   | 2a00:1450:401b:805::200e | 128 |
| MX     | 10 aspmx.example.com.    | 128 |

TTL for individual records of the same type is currently unsupported due to the necessity to change the record value format and increased gas cost. Setting `dns.ttl` instead of TTL for individual records is recommended due to higher gas efficiency.

## Authority responses

It is a common practice in DNS to have an authority of a subdomain delegated to a parent domain. This mechanism is not necessary for crypto domains because the cost of subdomain registration is comparable to setting records. In other words, configuring subdomain using the parent domain has no benefit and may result in even higher gas cost due to the necessity to store associated subdomain name to each record.

Therefore, authority configurations are not supported by crypto domains at the moment.

## Distributed Web records

Distributed Web (Dweb) records are designed to allow one to configure a domain for distributed websites protocols like IPFS or Swarm. Such records are stored in `dweb.*` namespace. Each protocol has its own sub-namespace for its data using canonic name. Example: Swarm's protocol canonic name is `bzz` so its records are stored at `dns.bzz.*` namespace.

Record structure can be different based on the protocol. However, all protocols have a common `.hash` record used to reference a content in the distributed network. Example: `dns.ipfs.hash` for IPFS protocol.

See Resolving Domains in a Browser section for an information how to interpret those records.

## Legacy Records Support

As of Q3 2020, most `.crypto` domains are configured using legacy record names for IPFS hash and redirect domain:

1. `ipfs.html.value` deprecated in favor of `dweb.ipfs.hash`
2. `ipfs.redirect_domain` deprecated in favor of `browser.redirect_url`

Browsers are strongly recommended to support those records as a fallback when corresponding replacement records are not set.

## Program Code

The following are various examples of implementing program code programmed using the Solidity programming language.

## IRegistry.sol

```
pragma solidity 0.5.12;

import "@openzeppelin/contracts/token/ERC721/IERC721Metadata.sol";

contract IRegistry is IERC721Metadata {

    event NewURI(uint256 indexed tokenId, string uri);

    event NewURIPrefix(string prefix);

    event Resolve(uint256 indexed tokenId, address indexed to);

    event Sync(address indexed resolver, uint256 indexed updateId, uint256 indexed tokenId);

    /**
     * @dev Controlled function to set the token URI Prefix for all tokens.
     * @param prefix string URI to assign
     */
    function controlledSetTokenURIPrefix(string calldata prefix) external;

    /**
     * @dev Returns whether the given spender can transfer a given token ID.
     * @param spender address of the spender to query
     * @param tokenId uint256 ID of the token to be transferred
     * @return bool whether the msg.sender is approved for the given token
ID,
     * is an operator of the owner, or is the owner of the token
     */
    function isApprovedOrOwner(address spender, uint256 tokenId) external
view returns (bool);

    /**
     * @dev Mints a new a child token.
     * Calculates child token ID using a namehash function.
     * Requires the msg.sender to be the owner, approved, or operator of
tokenId.
     * Requires the token not exist.
     * @param to address to receive the ownership of the given token ID
     * @param tokenId uint256 ID of the parent token
     * @param label subdomain label of the child token ID
     */
    function mintChild(address to, uint256 tokenId, string calldata label)
external;

    /**
     * @dev Controlled function to mint a given token ID.
     * Requires the msg.sender to be controller.
     * Requires the token ID to not exist.
     * @param to address the given token ID will be minted to
     * @param label string that is a subdomain
     * @param tokenId uint256 ID of the parent token
     */
}
```

```

function controlledMintChild(address to, uint256 tokenId, string calldata
label) external;

/**
 * @dev Transfers the ownership of a child token ID to another address.
 * Calculates child token ID using a namehash function.
 * Requires the msg.sender to be the owner, approved, or operator of
tokenId.
 * Requires the token already exist.
 * @param from current owner of the token
 * @param to address to receive the ownership of the given token ID
 * @param tokenId uint256 ID of the token to be transferred
 * @param label subdomain label of the child token ID
 */
function transferFromChild(address from, address to, uint256 tokenId,
string calldata label) external;

/**
 * @dev Controlled function to transfers the ownership of a token ID to
 * another address.
 * Requires the msg.sender to be controller.
 * Requires the token already exist.
 * @param from current owner of the token
 * @param to address to receive the ownership of the given token ID
 * @param tokenId uint256 ID of the token to be transferred
 */
function controlledTransferFrom(address from, address to, uint256
tokenId) external;

/**
 * @dev Safely transfers the ownership of a child token ID to another
address.
 * Calculates child token ID using a namehash function.
 * Implements a ERC721Receiver check unlike transferFromChild.
 * Requires the msg.sender to be the owner, approved, or operator of
tokenId.
 * Requires the token already exist.
 * @param from current owner of the token
 * @param to address to receive the ownership of the given token ID
 * @param tokenId uint256 parent ID of the token to be transferred
 * @param label subdomain label of the child token ID
 * @param _data bytes data to send along with a safe transfer check
 */
function safeTransferFromChild(address from, address to, uint256 tokenId,
string calldata label, bytes calldata _data) external;

/// Shorthand for calling the above ^^^ safeTransferFromChild function
with an empty _data parameter. Similar to ERC721.safeTransferFrom.
function safeTransferFromChild(address from, address to, uint256 tokenId,
string calldata label) external;

/**
 * @dev Controlled function to safely transfers the ownership of a token
ID
 * to another address.
 * Implements a ERC721Receiver check unlike controlledSafeTransferFrom.
 * Requires the msg.sender to be controller.

```

```

    * Requires the token already exist.
    * @param from current owner of the token
    * @param to address to receive the ownership of the given token ID
    * @param tokenId uint256 parent ID of the token to be transferred
    * @param _data bytes data to send along with a safe transfer check
    */
    function controlledSafeTransferFrom(address from, address to, uint256
tokenId, bytes calldata _data) external;

    /**
    * @dev Burns a child token ID.
    * Calculates child token ID using a namehash function.
    * Requires the msg.sender to be the owner, approved, or operator of
tokenId.
    * Requires the token already exist.
    * @param tokenId uint256 ID of the token to be transferred
    * @param label subdomain label of the child token ID
    */
    function burnChild(uint256 tokenId, string calldata label) external;

    /**
    * @dev Controlled function to burn a given token ID.
    * Requires the msg.sender to be controller.
    * Requires the token already exist.
    * @param tokenId uint256 ID of the token to be burned
    */
    function controlledBurn(uint256 tokenId) external;

    /**
    * @dev Sets the resolver of a given token ID to another address.
    * Requires the msg.sender to be the owner, approved, or operator.
    * @param to address the given token ID will resolve to
    * @param tokenId uint256 ID of the token to be transferred
    */
    function resolveTo(address to, uint256 tokenId) external;

    /**
    * @dev Gets the resolver of the specified token ID.
    * @param tokenId uint256 ID of the token to query the resolver of
    * @return address currently marked as the resolver of the given token ID
    */
    function resolverOf(uint256 tokenId) external view returns (address);

    /**
    * @dev Controlled function to sets the resolver of a given token ID.
    * Requires the msg.sender to be controller.
    * @param to address the given token ID will resolve to
    * @param tokenId uint256 ID of the token to be transferred
    */
    function controlledResolveTo(address to, uint256 tokenId) external;

    /**
    * @dev Provides child token (subdomain) of provided tokenId.
    * @param tokenId uint256 ID of the token
    * @param label of subdomain (for `aaa.bbb.crypto` it will be `aaa`)
    */

```

```

    function childIdOf(uint256 tokenId, string calldata label) external pure
returns (uint256);

    /**
    * @dev Transfer domain ownership without resetting domain records.
    * @param to address of new domain owner
    * @param tokenId uint256 ID of the token to be transferred
    */
    function setOwner(address to, uint256 tokenId) external;
}

```

## Registry.sol

```

pragma solidity 0.5.12;

import "../IRegistry.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721Burnable.sol";
import "../util/ControllerRole.sol";

// solium-disable no-empty-blocks,error-reason

/**
* @title Registry
* @dev An ERC721 Token see https://eips.ethereum.org/EIPS/eip-721. With
* additional functions so other trusted contracts to interact with the
tokens.
*/
contract Registry is IRegistry, ControllerRole, ERC721Burnable {

    // Optional mapping for token URIs
    mapping(uint256 => string) internal _tokenURIs;

    string internal _prefix;

    // Mapping from token ID to resolver address
    mapping (uint256 => address) internal _tokenResolvers;

    // uint256(keccak256(abi.encodePacked(uint256(0x0),
keccak256(abi.encodePacked("crypto")))))
    uint256 private constant _CRYPTO_HASH =
        0x0f4a10a4f46c288cea365fcf45cccf0e9d901b945b9829ccdb54c10dc3cb7a6f;

    modifier onlyApprovedOrOwner(uint256 tokenId) {
        require(_isApprovedOrOwner(msg.sender, tokenId));
        _;
    }

    constructor () public {
        _mint(address(0xdead), _CRYPTO_HASH);
        // register the supported interfaces to conform to ERC721 via ERC165
        _registerInterface(0x5b5e139f); // ERC721 Metadata Interface
        _tokenURIs[root()] = "crypto";
        emit NewURI(root(), "crypto");
    }
}

```

```

    /// ERC721 Metadata extension

    function name() external view returns (string memory) {
        return ".crypto";
    }

    function symbol() external view returns (string memory) {
        return "UD";
    }

    function tokenURI(uint256 tokenId) external view returns (string memory)
    {
        require(_exists(tokenId));
        return string(abi.encodePacked(_prefix, _tokenURIs[tokenId]));
    }

    function controlledSetTokenURIPrefix(string calldata prefix) external
    onlyController {
        _prefix = prefix;
        emit NewURIPrefix(prefix);
    }

    /// Ownership

    function isApprovedOrOwner(address spender, uint256 tokenId) external
    view returns (bool) {
        return _isApprovedOrOwner(spender, tokenId);
    }

    /// Registry Constants

    function root() public pure returns (uint256) {
        return _CRYPTO_HASH;
    }

    function childIdOf(uint256 tokenId, string calldata label) external pure
    returns (uint256) {
        return _childId(tokenId, label);
    }

    /// Minting

    function mintChild(address to, uint256 tokenId, string calldata label)
    external onlyApprovedOrOwner(tokenId) {
        _mintChild(to, tokenId, label);
    }

    function controlledMintChild(address to, uint256 tokenId, string calldata
    label) external onlyController {
        _mintChild(to, tokenId, label);
    }

    function safeMintChild(address to, uint256 tokenId, string calldata
    label) external onlyApprovedOrOwner(tokenId) {
        _safeMintChild(to, tokenId, label, "");
    }

```

```

    function safeMintChild(address to, uint256 tokenId, string calldata
label, bytes calldata _data)
        external
        onlyApprovedOrOwner(tokenId)
    {
        _safeMintChild(to, tokenId, label, _data);
    }

    function controlledSafeMintChild(address to, uint256 tokenId, string
calldata label, bytes calldata _data)
        external
        onlyController
    {
        _safeMintChild(to, tokenId, label, _data);
    }

    /// Transferring

    function setOwner(address to, uint256 tokenId) external
onlyApprovedOrOwner(tokenId) {
        super._transferFrom(ownerOf(tokenId), to, tokenId);
    }

    function transferFromChild(address from, address to, uint256 tokenId,
string calldata label)
        external
        onlyApprovedOrOwner(tokenId)
    {
        _transferFrom(from, to, _childId(tokenId, label));
    }

    function controlledTransferFrom(address from, address to, uint256
tokenId) external onlyController {
        _transferFrom(from, to, tokenId);
    }

    function safeTransferFromChild(
        address from,
        address to,
        uint256 tokenId,
        string memory label,
        bytes memory _data
    ) public onlyApprovedOrOwner(tokenId) {
        uint256 childId = _childId(tokenId, label);
        _transferFrom(from, to, childId);
        require(_checkOnERC721Received(from, to, childId, _data));
    }

    function safeTransferFromChild(address from, address to, uint256 tokenId,
string calldata label) external {
        safeTransferFromChild(from, to, tokenId, label, "");
    }

    function controlledSafeTransferFrom(address from, address to, uint256
tokenId, bytes calldata _data)
        external

```

```

        onlyController
    {
        _transferFrom(from, to, tokenId);
        require(_checkOnERC721Received(from, to, tokenId, _data));
    }

    /// Burning

    function burnChild(uint256 tokenId, string calldata label) external
    onlyApprovedOrOwner(tokenId) {
        _burn(_childId(tokenId, label));
    }

    function controlledBurn(uint256 tokenId) external onlyController {
        _burn(tokenId);
    }

    /// Resolution

    function resolverOf(uint256 tokenId) external view returns (address) {
        address resolver = _tokenResolvers[tokenId];
        require(resolver != address(0));
        return resolver;
    }

    function resolveTo(address to, uint256 tokenId) external
    onlyApprovedOrOwner(tokenId) {
        _resolveTo(to, tokenId);
    }

    function controlledResolveTo(address to, uint256 tokenId) external
    onlyController {
        _resolveTo(to, tokenId);
    }

    function sync(uint256 tokenId, uint256 updateId) external {
        require(_tokenResolvers[tokenId] == msg.sender);
        emit Sync(msg.sender, updateId, tokenId);
    }

    /// Internal

    function _childId(uint256 tokenId, string memory label) internal pure
    returns (uint256) {
        require(bytes(label).length != 0);
        return uint256(keccak256(abi.encodePacked(tokenId,
        keccak256(abi.encodePacked(label))))));
    }

    function _mintChild(address to, uint256 tokenId, string memory label)
    internal {
        uint256 childId = _childId(tokenId, label);
        _mint(to, childId);

        require(bytes(label).length != 0);
        require(!_exists(childId));
    }

```

```

        bytes memory domain = abi.encodePacked(label, ".",
_tokenURIs[tokenId]);

        _tokenURIs[childId] = string(domain);
        emit NewURI(childId, string(domain));
    }

    function _safeMintChild(address to, uint256 tokenId, string memory label,
bytes memory _data) internal {
        _mintChild(to, tokenId, label);
        require(_checkOnERC721Received(address(0), to, _childId(tokenId,
label), _data));
    }

    function _transferFrom(address from, address to, uint256 tokenId)
internal {
        super._transferFrom(from, to, tokenId);
        // Clear resolver (if any)
        if (_tokenResolvers[tokenId] != address(0x0)) {
            delete _tokenResolvers[tokenId];
        }
    }

    function _burn(uint256 tokenId) internal {
        super._burn(tokenId);
        // Clear resolver (if any)
        if (_tokenResolvers[tokenId] != address(0x0)) {
            delete _tokenResolvers[tokenId];
        }
        // Clear metadata (if any)
        if (bytes(_tokenURIs[tokenId]).length != 0) {
            delete _tokenURIs[tokenId];
        }
    }

    function _resolveTo(address to, uint256 tokenId) internal {
        require(_exists(tokenId));
        emit Resolve(tokenId, to);
        _tokenResolvers[tokenId] = to;
    }
}

```

## IResolver.sol

```

pragma solidity 0.5.12;
pragma experimental ABIEncoderV2;

contract IResolver {
    /**
     * @dev Reset all domain records and set new ones
     * @param keys New record keys
     * @param values New record values
     * @param tokenId ERC-721 token id of the domain

```

```

    */
    function reconfigure(string[] memory keys, string[] memory values,
uint256 tokenId) public;

    /**
    * @dev Set or update domain records
    * @param keys New record keys
    * @param values New record values
    * @param tokenId ERC-721 token id of the domain
    */
    function setMany(string[] memory keys, string[] memory values, uint256
tokenId) public;

    /**
    * @dev Function to set record.
    * @param key The key set the value of.
    * @param value The value to set key to.
    * @param tokenId ERC-721 token id to set.
    */
    function set(string calldata key, string calldata value, uint256 tokenId)
external;

    /**
    * @dev Function to reset all existing records on a domain.
    * @param tokenId ERC-721 token id to set.
    */
    function reset(uint256 tokenId) external;
}

```

## Resolver.sol

```

pragma solidity 0.5.12;
pragma experimental ABIEncoderV2;

import './Registry.sol';
import './util/SignatureUtil.sol';
import './controllers/MintingController.sol';
import './IResolver.sol';
import './IResolverReader.sol';
// solium-disable error-reason

contract Resolver is IResolverReader, SignatureUtil, IResolver {

    event Set(uint256 indexed tokenId, string indexed keyIndex, string
indexed valueIndex, string key, string value);
    event NewKey(uint256 indexed tokenId, string indexed keyIndex, string
key);
    event ResetRecords(uint256 indexed tokenId);

    // Mapping from token ID to preset id to key to value
    mapping (uint256 => mapping (uint256 => mapping (string => string)))
internal _records;

    // Mapping from token ID to current preset id

```

```

mapping (uint256 => uint256) _tokenPresets;

// All keys that were set
mapping (uint256 => string) _hashedKeys;

MintingController internal _mintingController;

constructor(Registry, MintingController mintingController) public
SignatureUtil(registry) {
    require(address(registry) == mintingController.registry());
    _mintingController = mintingController;
}

/**
 * @dev Throws if called when not the resolver.
 */
modifier whenResolver(uint256 tokenId) {
    require(address(this) == _registry.resolverOf(tokenId),
"RESOLVER_DETACHED_FROM_DOMAIN");
    _;
}

modifier whenApprovedOrOwner(uint256 tokenId) {
    require(_registry.isApprovedOrOwner(msg.sender, tokenId),
"SENDER_IS_NOT_APPROVED_OR_OWNER");
    _;
}

function reset(uint256 tokenId) external whenApprovedOrOwner(tokenId) {
    _setPreset(now, tokenId);
}

function resetFor(uint256 tokenId, bytes calldata signature) external {
    _validate(keccak256(abi.encodeWithSelector(this.reset.selector,
tokenId)), tokenId, signature);
    _setPreset(now, tokenId);
}

/**
 * @dev Function to get record.
 * @param key The key to query the value of.
 * @param tokenId The token id to fetch.
 * @return The value string.
 */
function get(string memory key, uint256 tokenId) public view
whenResolver(tokenId) returns (string memory) {
    return _records[tokenId][_tokenPresets[tokenId]][key];
}

/**
 * @dev Function to get key by provided hash. Keys hashes can be found in
Sync event emitted by Registry.sol contract.
 * @param keyHash The key to query the value of.
 * @return The key string.
 */
function hashToKey(uint256 keyHash) public view returns (string memory) {
    return _hashedKeys[keyHash];
}

```

```

    }

    /**
     * @dev Function to get keys by provided key hashes. Keys hashes can be
     found in Sync event emitted by Registry.sol contract.
     * @param hashes The key to query the value of.
     * @return Keys
     */
    function hashesToKeys(uint256[] memory hashes) public view returns
(string[] memory) {
        uint256 keyCount = hashes.length;
        string[] memory values = new string[](keyCount);
        for (uint256 i = 0; i < keyCount; i++) {
            values[i] = hashToKey(hashes[i]);
        }

        return values;
    }

    /**
     * @dev Function get value by provied key hash. Keys hashes can be found
     in Sync event emitted by Registry.sol contract.
     * @param keyHash The key to query the value of.
     * @param tokenId The token id to set.
     * @return Key and value.
     */
    function getByHash(uint256 keyHash, uint256 tokenId) public view
whenResolver(tokenId) returns (string memory key, string memory value) {
        key = hashToKey(keyHash);
        value = get(key, tokenId);
    }

    /**
     * @dev Function get values by provied key hashes. Keys hashes can be
     found in Sync event emitted by Registry.sol contract.
     * @param keyHashes The key to query the value of.
     * @param tokenId The token id to set.
     * @return Keys and values.
     */
    function getManyByHash(
        uint256[] memory keyHashes,
        uint256 tokenId
    ) public view whenResolver(tokenId) returns (string[] memory keys,
string[] memory values) {
        uint256 keyCount = keyHashes.length;
        keys = new string[](keyCount);
        values = new string[](keyCount);
        for (uint256 i = 0; i < keyCount; i++) {
            (keys[i], values[i]) = getByHash(keyHashes[i], tokenId);
        }
    }

    function preconfigure(
        string[] memory keys,
        string[] memory values,
        uint256 tokenId
    ) public {

```

```

        require(_mintingController.isMinter(msg.sender),
"SENDER_IS_NOT_MINTER");
        _setMany(_tokenPresets[tokenId], keys, values, tokenId);
    }

    /**
     * @dev Function to set record.
     * @param key The key set the value of.
     * @param value The value to set key to.
     * @param tokenId The token id to set.
     */
    function set(string calldata key, string calldata value, uint256 tokenId)
external whenApprovedOrOwner(tokenId) {
        _set(_tokenPresets[tokenId], key, value, tokenId);
    }

    /**
     * @dev Function to set record on behalf of an address.
     * @param key The key set the value of.
     * @param value The value to set key to.
     * @param tokenId The token id to set.
     * @param signature The signature to verify the transaction with.
     */
    function setFor(
        string calldata key,
        string calldata value,
        uint256 tokenId,
        bytes calldata signature
    ) external {
        _validate(keccak256(abi.encodeWithSelector(this.set.selector, key,
value, tokenId)), tokenId, signature);
        _set(_tokenPresets[tokenId], key, value, tokenId);
    }

    /**
     * @dev Function to get multiple record.
     * @param keys The keys to query the value of.
     * @param tokenId The token id to fetch.
     * @return The values.
     */
    function getMany(string[] calldata keys, uint256 tokenId) external view
whenResolver(tokenId) returns (string[] memory) {
        uint256 keyCount = keys.length;
        string[] memory values = new string[](keyCount);
        uint256 preset = _tokenPresets[tokenId];
        for (uint256 i = 0; i < keyCount; i++) {
            values[i] = _records[tokenId][preset][keys[i]];
        }
        return values;
    }

    function setMany(
        string[] memory keys,
        string[] memory values,
        uint256 tokenId
    ) public whenApprovedOrOwner(tokenId) {
        _setMany(_tokenPresets[tokenId], keys, values, tokenId);
    }

```

```

    }

    /**
     * @dev Function to set record on behalf of an address.
     * @param keys The keys set the values of.
     * @param values The values to set keys to.
     * @param tokenId The token id to set.
     * @param signature The signature to verify the transaction with.
     */
    function setManyFor(
        string[] memory keys,
        string[] memory values,
        uint256 tokenId,
        bytes memory signature
    ) public {
        _validate(keccak256(abi.encodeWithSelector(this.setMany.selector,
keys, values, tokenId)), tokenId, signature);
        _setMany(_tokenPresets[tokenId], keys, values, tokenId);
    }

    /**
     * @dev Function to reset all domain records and set new ones.
     * @param keys records keys.
     * @param values records values.
     * @param tokenId domain token id.
     */
    function reconfigure(string[] memory keys, string[] memory values,
uint256 tokenId) public whenApprovedOrOwner(tokenId) {
        _reconfigure(keys, values, tokenId);
    }

    /**
     * @dev Delegated version of reconfigure() function.
     * @param keys records keys.
     * @param values records values.
     * @param tokenId domain token id.
     * @param signature user signature.
     */
    function reconfigureFor(
        string[] memory keys,
        string[] memory values,
        uint256 tokenId,
        bytes memory signature
    ) public {
        _validate(keccak256(abi.encodeWithSelector(this.reconfigure.selector,
keys, values, tokenId)), tokenId, signature);
        _reconfigure(keys, values, tokenId);
    }

    // reset records
    function _setPreset(uint256 presetId, uint256 tokenId) internal {
        _tokenPresets[tokenId] = presetId;
        _registry.sync(tokenId, 0); // notify registry that domain records
were reset
        emit ResetRecords(tokenId);
    }

```

```

/**
 * @dev Internal function to set record. As opposed to set, this imposes
no restrictions on msg.sender.
 * @param preset to set key/values on
 * @param key of record to be set
 * @param value of record to be set
 * @param tokenId uint256 ID of the token
 */
function _set(uint256 preset, string memory key, string memory value,
uint256 tokenId) internal {
    uint256 keyHash = uint256(keccak256(bytes(key)));
    bool isNewKey = bytes(_records[tokenId][preset][key]).length == 0;
    _registry.sync(tokenId, keyHash);
    _records[tokenId][preset][key] = value;

    if (bytes(_hashedKeys[keyHash]).length == 0) {
        _hashedKeys[keyHash] = key;
    }

    if (isNewKey) {
        emit NewKey(tokenId, key, key);
    }
    emit Set(tokenId, key, value, key, value);
}

/**
 * @dev Internal function to set multiple records. As opposed to setMany,
this imposes
 * no restrictions on msg.sender.
 * @param preset to set key/values on
 * @param keys of record to be set
 * @param values of record to be set
 * @param tokenId uint256 ID of the token
 */
function _setMany(uint256 preset, string[] memory keys, string[] memory
values, uint256 tokenId) internal {
    uint256 keyCount = keys.length;
    for (uint256 i = 0; i < keyCount; i++) {
        _set(preset, keys[i], values[i], tokenId);
    }
}

/**
 * @dev Internal function to reset all domain records and set new ones.
 * @param keys records keys.
 * @param values records values.
 * @param tokenId domain token id.
 */
function _reconfigure(string[] memory keys, string[] memory values,
uint256 tokenId) internal {
    _setPreset(now, tokenId);
    _setMany(_tokenPresets[tokenId], keys, values, tokenId);
}
}

```

## IDataReader.sol

```
pragma solidity 0.5.12;
pragma experimental ABIEncoderV2;

interface IDataReader {
    /**
     * @dev Function to get resolver address, owner address and requested
records.
     * @param keys The keys to query the value of.
     * @param tokenId The token id to fetch.
     * @return Resolver address, owner address and values.
     */
    function getData(string[] calldata keys, uint256 tokenId)
        external
        view
        returns (
            address resolver,
            address owner,
            string[] memory values
        );

    /**
     * @dev Function to get resolver address, owner address and requested
records.
     * @param keyHashes Key hashes to query the value of.
     * @param tokenId The token id to fetch.
     * @return Resolver address, owner address, keys and values.
     */
    function getDataByHash(uint256[] calldata keyHashes, uint256 tokenId)
        external
        view
        returns (
            address resolver,
            address owner,
            string[] memory keys,
            string[] memory values
        );
}
```

## IResolverReader.sol

```
pragma solidity 0.5.12;
pragma experimental ABIEncoderV2;

interface IResolverReader {
    /**
     * @dev Gets the nonce of the specified address.
     * @param tokenId token ID for nonce query
     * @return nonce of the given address
     */
    function nonceOf(uint256 tokenId) external view returns (uint256);
}
```

```

/**
 * @return registry address
 */
function registry() external view returns (address);

/**
 * @dev Function to get record.
 * @param key The key to query the value of.
 * @param tokenId The token id to fetch.
 * @return The value string.
 */
function get(string calldata key, uint256 tokenId)
    external
    view
    returns (string memory);

/**
 * @dev Function to get multiple record.
 * @param keys The keys to query the value of.
 * @param tokenId The token id to fetch.
 * @return The values.
 */
function getMany(string[] calldata keys, uint256 tokenId)
    external
    view
    returns (string[] memory);

/**
 * @dev Function get value by provied key hash. Keys hashes can be found
in Sync event emitted by Registry.sol contract.
 * @param keyHash The key to query the value of.
 * @param tokenId The token id to set.
 * @return Key and value.
 */
function getByHash(uint256 keyHash, uint256 tokenId)
    external
    view
    returns (string memory key, string memory value);

/**
 * @dev Function get values by provied key hashes. Keys hashes can be
found in Sync event emitted by Registry.sol contract.
 * @param keyHashes The key to query the value of.
 * @param tokenId The token id to set.
 * @return Keys and values.
 */
function getManyByHash(uint256[] calldata keyHashes, uint256 tokenId)
    external
    view
    returns (string[] memory keys, string[] memory values);
}

```

## BulkWhitelistedRole.sol

```

pragma solidity 0.5.12;

import "@openzeppelin/contracts/access/Roles.sol";
import "@openzeppelin/contracts/access/roles/WhitelistedRole.sol";

/**
 * @title BulkWhitelistedRole
 * @dev a Whitelist role defined using the Open Zeppelin Role system with the
addition of bulkAddWhitelisted and
 * bulkRemoveWhitelisted.
 * @dev Whitelisted accounts have been approved by a WhitelistAdmin to
perform certain actions (e.g. participate in a
 * crowdsale). This role is special in that the only accounts that can add it
are WhitelistAdmins (who can also remove
 * it), and not Whitelisted themselves.
 */
contract BulkWhitelistedRole is WhitelistedRole {

    function bulkAddWhitelisted(address[] memory accounts) public
onlyWhitelistAdmin {
        for (uint256 index = 0; index < accounts.length; index++) {
            _addWhitelisted(accounts[index]);
        }
    }

    function bulkRemoveWhitelisted(address[] memory accounts) public
onlyWhitelistAdmin {
        for (uint256 index = 0; index < accounts.length; index++) {
            _removeWhitelisted(accounts[index]);
        }
    }
}

```

## ControllerRole.sol

```

pragma solidity 0.5.12;

import "@openzeppelin/contracts/access/Roles.sol";

// solium-disable error-reason

/**
 * @title ControllerRole
 * @dev An Controller role defined using the Open Zeppelin Role system.
 */
contract ControllerRole {

    using Roles for Roles.Role;

    // NOTE: Commented out standard Role events to save gas.
    // event ControllerAdded(address indexed account);
    // event ControllerRemoved(address indexed account);

```

```

Roles.Role private _controllers;

constructor () public {
    _addController(msg.sender);
}

modifier onlyController() {
    require(isController(msg.sender));
    _;
}

function isController(address account) public view returns (bool) {
    return _controllers.has(account);
}

function addController(address account) public onlyController {
    _addController(account);
}

function renounceController() public {
    _removeController(msg.sender);
}

function _addController(address account) internal {
    _controllers.add(account);
    // emit ControllerAdded(account);
}

function _removeController(address account) internal {
    _controllers.remove(account);
    // emit ControllerRemoved(account);
}
}

```

## SignatureUtil.sol

```

pragma solidity 0.5.12;

import '../Registry.sol';
import '@openzeppelin/contracts/cryptography/ECDSA.sol';

// solium-disable error-reason

contract SignatureUtil {
    using ECDSA for bytes32;

    // Mapping from owner to a nonce
    mapping (uint256 => uint256) internal _nonces;

    Registry internal _registry;

    constructor(Registry registry) public {
        _registry = registry;
    }
}

```

```

    }

    function registry() external view returns (address) {
        return address(_registry);
    }

    /**
     * @dev Gets the nonce of the specified address.
     * @param tokenId token ID for nonce query
     * @return nonce of the given address
     */
    function nonceOf(uint256 tokenId) external view returns (uint256) {
        return _nonces[tokenId];
    }

    function _validate(bytes32 hash, uint256 tokenId, bytes memory signature)
    internal {
        uint256 nonce = _nonces[tokenId];

        address signer = keccak256(abi.encodePacked(hash, address(this),
        nonce)).toEthSignedMessageHash().recover(signature);
        require(
            signer != address(0) &&
            _registry.isApprovedOrOwner(
                signer,
                tokenId
            ),
            "INVALID_SIGNATURE"
        );

        _nonces[tokenId] += 1;
    }
}

```

## WhitelistedMinter.sol

```

pragma solidity 0.5.12;
pragma experimental ABIEncoderV2;

import '@openzeppelin/contracts/cryptography/ECDSA.sol';
import "../BulkWhitelistedRole.sol";
import "../controllers/IMintingController.sol";
import "../controllers/MintingController.sol";
import "../Registry.sol";
import "../Resolver.sol";

/**
 * @title WhitelistedMinter
 * @dev Defines the functions for distribution of Second Level Domains
 * (SLD)s.
 */
contract WhitelistedMinter is IMintingController, BulkWhitelistedRole {
    using ECDSA for bytes32;

```

```

    event Relayed(address indexed sender, address indexed signer, bytes4
indexed funcSig, bytes32 dataHash);

    string public constant NAME = 'Unstoppable Whitelisted Minter';
    string public constant VERSION = '0.3.0';

    MintingController internal _mintingController;
    Resolver internal _resolver;
    Registry internal _registry;

    /*
    * bytes4(keccak256('mintSLD(address,string)')) == 0x4c0b0ed2
    */
    bytes4 private constant _SIG_MINT = 0x4c0b0ed2;

    /*
    *
    bytes4(keccak256('mintSLDToDefaultResolver(address,string,string[],string[])'
)) == 0x3d7989fe
    */
    bytes4 private constant _SIG_MINT_DEF_RESOLVER = 0x3d7989fe;

    /*
    *
    bytes4(keccak256('mintSLDToResolver(address,string,string[],string[],address)
')) == 0xaceb4764
    */
    bytes4 private constant _SIG_MINT_RESOLVER = 0xaceb4764;

    /*
    * bytes4(keccak256('safeMintSLD(address,string)')) == 0xb2da2979
    */
    bytes4 private constant _SIG_SAFE_MINT = 0xb2da2979;

    /*
    * bytes4(keccak256('safeMintSLD(address,string,bytes)')) == 0xbe362e2e
    */
    bytes4 private constant _SIG_SAFE_MINT_DATA = 0xbe362e2e;

    /*
    *
    bytes4(keccak256('safeMintSLDToDefaultResolver(address,string,string[],string
[])')) == 0x61050ffd
    */
    bytes4 private constant _SIG_SAFE_MINT_DEF_RESOLVER = 0x61050ffd;

    /*
    *
    bytes4(keccak256('safeMintSLDToDefaultResolver(address,string,string[],string
[],bytes)')) == 0x4b18abea
    */
    bytes4 private constant _SIG_SAFE_MINT_DEF_RESOLVER_DATA = 0x4b18abea;

    /*

```

```

    *
    bytes4(keccak256('safeMintSLDToResolver(address,string,string[],string[],address)')) == 0x4b44c01a
    */
    bytes4 private constant _SIG_SAFE_MINT_RESOLVER = 0x4b44c01a;

    /*
    *
    bytes4(keccak256('safeMintSLDToResolver(address,string,string[],string[],bytes,address)')) == 0x898851f8
    */
    bytes4 private constant _SIG_SAFE_MINT_RESOLVER_DATA = 0x898851f8;

    constructor(MintingController mintingController) public {
        _mintingController = mintingController;
        _registry = Registry(mintingController.registry());
        _addWhitelisted(address(this));
    }

    function renounceMinter() external onlyWhitelistAdmin {
        _mintingController.renounceMinter();
    }

    /**
     * Renounce whitelisted account with funds' forwarding
     */
    function closeWhitelisted(address payable receiver)
        external
        payable
        onlyWhitelisted
    {
        require(receiver != address(0x0), "WhitelistedMinter:
RECEIVER_IS_EMPTY");

        renounceWhitelisted();
        receiver.transfer(msg.value);
    }

    /**
     * Replace whitelisted account by new account with funds' forwarding
     */
    function rotateWhitelisted(address payable receiver)
        external
        payable
        onlyWhitelisted
    {
        require(receiver != address(0x0), "WhitelistedMinter:
RECEIVER_IS_EMPTY");

        _addWhitelisted(receiver);
        renounceWhitelisted();
        receiver.transfer(msg.value);
    }

    function mintSLD(address to, string calldata label)
        external
        onlyWhitelisted

```

```

{
    _mintingController.mintSLD(to, label);
}

function safeMintSLD(address to, string calldata label)
    external
    onlyWhitelisted
{
    _mintingController.safeMintSLD(to, label);
}

function safeMintSLD(
    address to,
    string calldata label,
    bytes calldata _data
) external onlyWhitelisted {
    _mintingController.safeMintSLD(to, label, _data);
}

function mintSLDToDefaultResolver(
    address to,
    string memory label,
    string[] memory keys,
    string[] memory values
) public onlyWhitelisted {
    mintSLDToResolver(to, label, keys, values, address(_resolver));
}

function mintSLDToResolver(
    address to,
    string memory label,
    string[] memory keys,
    string[] memory values,
    address resolver
) public onlyWhitelisted {
    _mintingController.mintSLDWithResolver(to, label, resolver);
    preconfigureResolver(label, keys, values, resolver);
}

function safeMintSLDToDefaultResolver(
    address to,
    string memory label,
    string[] memory keys,
    string[] memory values
) public onlyWhitelisted {
    safeMintSLDToResolver(to, label, keys, values, address(_resolver));
}

function safeMintSLDToResolver(
    address to,
    string memory label,
    string[] memory keys,
    string[] memory values,
    address resolver
) public onlyWhitelisted {
    _mintingController.safeMintSLDWithResolver(to, label, resolver);
    preconfigureResolver(label, keys, values, resolver);
}

```

```

    }

    function safeMintSLDToDefaultResolver(
        address to,
        string memory label,
        string[] memory keys,
        string[] memory values,
        bytes memory _data
    ) public onlyWhitelisted {
        safeMintSLDToResolver(to, label, keys, values, _data,
address(_resolver));
    }

    function safeMintSLDToResolver(
        address to,
        string memory label,
        string[] memory keys,
        string[] memory values,
        bytes memory _data,
        address resolver
    ) public onlyWhitelisted {
        _mintingController.safeMintSLDWithResolver(to, label, resolver,
_data);
        preconfigureResolver(label, keys, values, resolver);
    }

    function setDefaultResolver(address resolver) external onlyWhitelistAdmin
{
    _resolver = Resolver(resolver);
}

    function getDefaultResolver() external view returns (address) {
        return address(_resolver);
    }

    /**
     * Relay allows execute transaction on behalf of whitelisted minter.
     * The function verify signature of call data parameter before execution.
     * It allows anybody send transaction on-chain when minter has provided
proper parameters.
     * The function allows to relaying calls of fixed functions. The
restriction defined in function `verifyCall`
     */
    function relay(bytes calldata data, bytes calldata signature) external
returns(bytes memory) {
        bytes32 dataHash = keccak256(data);
        address signer = verifySigner(dataHash, signature);
        bytes memory _data = data;
        bytes4 funcSig = verifyCall(_data);

        /* solium-disable-next-line security/no-low-level-calls */
        (bool success, bytes memory result) = address(this).call(data);
        if (success == false) {
            /* solium-disable-next-line security/no-inline-assembly */
            assembly {
                let ptr := mload(0x40)
                let size := returndatasize

```

```

        returndatacopy(ptr, 0, size)
        revert(ptr, size)
    }
}

emit Relayed(msg.sender, signer, funcSig, dataHash);
return result;
}

function preconfigureResolver(
    string memory label,
    string[] memory keys,
    string[] memory values,
    address resolver
) private {
    if(keys.length == 0) {
        return;
    }

    Resolver(resolver).preconfigure(keys, values,
_registry.childIdOf(_registry.root(), label));
}

function verifySigner(bytes32 data, bytes memory signature) private view
returns(address signer) {
    signer = keccak256(abi.encodePacked(data, address(this)))
        .toEthSignedMessageHash()
        .recover(signature);
    require(signer != address(0), 'WhitelistedMinter:
SIGNATURE_IS_INVALID');
    require(isWhitelisted(signer), 'WhitelistedMinter:
SIGNER_IS_NOT_WHITELISTED');
}

function verifyCall(bytes memory data) private pure returns(bytes4 sig) {
    /* solium-disable-next-line security/no-inline-assembly */
    assembly {
        sig := mload(add(data, add(0x20, 0)))
    }

    bool isSupported = sig == _SIG_MINT ||
        sig == _SIG_MINT_DEF_RESOLVER ||
        sig == _SIG_MINT_RESOLVER ||
        sig == _SIG_SAFE_MINT ||
        sig == _SIG_SAFE_MINT_DATA ||
        sig == _SIG_SAFE_MINT_DEF_RESOLVER ||
        sig == _SIG_SAFE_MINT_DEF_RESOLVER_DATA ||
        sig == _SIG_SAFE_MINT_RESOLVER ||
        sig == _SIG_SAFE_MINT_RESOLVER_DATA;

    require(isSupported, 'WhitelistedMinter: UNSUPPORTED_CALL');
}
}

```

## IMintingController.sol

```
pragma solidity 0.5.12;

interface IMintingController {

    /**
     * @dev Minter function that mints a Second Level Domain (SLD).
     * @param to address to mint the new SLD to.
     * @param label SLD label to mint.
     */
    function mintSLD(address to, string calldata label) external;

    /**
     * @dev Minter function that safely mints a Second Level Domain (SLD).
     * Implements a ERC721Receiver check unlike mintSLD.
     * @param to address to mint the new SLD to.
     * @param label SLD label to mint.
     */
    function safeMintSLD(address to, string calldata label) external;

    /**
     * @dev Minter function that safely mints a Second Level Domain (SLD).
     * Implements a ERC721Receiver check unlike mintSLD.
     * @param to address to mint the new SLD to.
     * @param label SLD label to mint.
     * @param _data bytes data to send along with a safe transfer check
     */
    function safeMintSLD(address to, string calldata label, bytes calldata
_data) external;
}
```

## MintingController.sol

```
pragma solidity 0.5.12;

import "@openzeppelin/contracts/access/roles/MinterRole.sol";
import "../IMintingController.sol";
import "../Registry.sol";

/**
 * @title MintingController
 * @dev Defines the functions for distribution of Second Level Domains
 (SLD)s.
 */
contract MintingController is IMintingController, MinterRole {

    Registry internal _registry;

    constructor (Registry registry) public {
        _registry = registry;
    }
}
```

```

function registry() external view returns (address) {
    return address(_registry);
}

function mintSLD(address to, string memory label) public onlyMinter {
    _registry.controlledMintChild(to, _registry.root(), label);
}

function safeMintSLD(address to, string calldata label) external {
    safeMintSLD(to, label, "");
}

function safeMintSLD(address to, string memory label, bytes memory _data)
public onlyMinter {
    _registry.controlledSafeMintChild(to, _registry.root(), label,
_data);
}

function mintSLDWithResolver(address to, string memory label, address
resolver) public onlyMinter {
    _registry.controlledMintChild(to, _registry.root(), label);
    _registry.controlledResolveTo(resolver,
_registry.childIdOf(_registry.root(), label));
}

function safeMintSLDWithResolver(address to, string calldata label,
address resolver) external {
    safeMintSLD(to, label, "");
    _registry.controlledResolveTo(resolver,
_registry.childIdOf(_registry.root(), label));
}

function safeMintSLDWithResolver(address to, string calldata label,
address resolver, bytes calldata _data) external {
    safeMintSLD(to, label, _data);
    _registry.controlledResolveTo(resolver,
_registry.childIdOf(_registry.root(), label));
}
}

```

## ISignatureController.sol

```

pragma solidity 0.5.12;

/**
 * @title ISignatureController
 * @dev All of the signature controller functions follow the same pattern.
Each
 * function mirrors another function in another controller or on the
registry.
 * The intent is that these signature functions can be an alternative to
signing

```

```

* away unilateral control via an approval without having to build
significant
* smart contract infrastructure, and without the domain owner having to have
* ether for gas.
*
* @dev The way these functions work is by validating a extra signature
argument
* of all of the arguments of the function and a nonce, and crosschecking the
* recovered address with the registry to ensure that the signer is the
owner.
* Instead of checking msg.sender for authentication.
*
* @dev The method we use for signing is non standard. It'd be worth it to
* consider using EIP712 Typed signing https://eips.ethereum.org/EIPS/eip-
712.
*/
interface ISignatureController {

    /**
    * @dev Returns the given owners' nonce.
    * @param tokenId token ID to query the nonce of
    * @return uint256 nonce of the owner
    */
    function nonceOf(uint256 tokenId) external view returns (uint256);

    /// A signature function based on transferFrom inside Open Zeppelin's
    ERC721.sol.
    function transferFromFor(address from, address to, uint256 tokenId, bytes
    calldata signature) external;

    /// A signature function based on safeTransferFrom inside Open Zeppelin's
    ERC721.sol.
    function safeTransferFromFor(address from, address to, uint256 tokenId,
    bytes calldata _data, bytes calldata signature) external;
    function safeTransferFromFor(address from, address to, uint256 tokenId,
    bytes calldata signature) external;

    /// A signature function based on resolveTo inside ./IRegistry.sol.
    function resolveToFor(address to, uint256 tokenId, bytes calldata
    signature) external;

    /// A signature function based on burn inside ./IRegistry.sol.
    function burnFor(uint256 tokenId, bytes calldata signature) external;

    /// A signature function based on mintChild inside
    ./IChildController.sol.
    function mintChildFor(address to, uint256 tokenId, string calldata label,
    bytes calldata signature) external;

    /// A signature function based on transferFromChild inside
    ./IChildController.sol.
    function transferFromChildFor(address from, address to, uint256 tokenId,
    string calldata label, bytes calldata signature) external;

    /// A signature function based on safeTransferFromChild inside
    ./IChildController.sol.

```

```

        function safeTransferFromChildFor(address from, address to, uint256
tokenId, string calldata label, bytes calldata _data, bytes calldata
signature) external;
        function safeTransferFromChildFor(address from, address to, uint256
tokenId, string calldata label, bytes calldata signature) external;

        /// A signature function based on burnChild inside
        ./IChildController.sol.
        function burnChildFor(uint256 tokenId, string calldata label, bytes
calldata signature) external;
    }

```

## SignatureController.sol

```

pragma solidity 0.5.12;

import "./ISignatureController.sol";
import '@openzeppelin/contracts/cryptography/ECDSA.sol';
import "../Registry.sol";
import "../util/SignatureUtil.sol";

// solium-disable error-reason

/**
 * @title SignatureController
 * @dev The SignatureController allows any account to submit select
management
 * transactions on behalf of a token owner.
 */
contract SignatureController is ISignatureController, SignatureUtil {

    constructor (Registry registry) public SignatureUtil(registry) {}

    /*
     * 0x23b872dd ==
bytes4(keccak256('transferFrom(address,address,uint256)'))
     */
    function transferFromFor(address from, address to, uint256 tokenId, bytes
calldata signature) external {
        _validate(
            keccak256(abi.encodeWithSelector(0x23b872dd, from, to, tokenId)),
            tokenId,
            signature
        );
        _registry.controlledTransferFrom(from, to, tokenId);
    }

    /*
     * 0xb88d4fde ==
bytes4(keccak256('safeTransferFrom(address,address,uint256,bytes)'))
     */
    function safeTransferFromFor(
        address from,

```

```

        address to,
        uint256 tokenId,
        bytes calldata _data,
        bytes calldata signature
    )
    external
    {
        _validate(
            keccak256(abi.encodeWithSelector(0xb88d4fde, from, to, tokenId,
_data)),
            tokenId,
            signature
        );
        _registry.controlledSafeTransferFrom(from, to, tokenId, _data);
    }

    /*
    * 0x42842e0e ==
bytes4(keccak256('safeTransferFrom(address,address,uint256)'))
    */
    function safeTransferFromFor(address from, address to, uint256 tokenId,
bytes calldata signature) external {
        _validate(
            keccak256(abi.encodeWithSelector(0x42842e0e, from, to, tokenId)),
            tokenId,
            signature
        );
        _registry.controlledSafeTransferFrom(from, to, tokenId, "");
    }

    /*
    * 0x42966c68 == bytes4(keccak256('burn(uint256)'))
    */
    function burnFor(uint256 tokenId, bytes calldata signature) external {
        _validate(
            keccak256(abi.encodeWithSelector(0x42966c68, tokenId)),
            tokenId,
            signature
        );
        _registry.controlledBurn(tokenId);
    }

    /*
    * 0xd8d3cc6e == bytes4(keccak256('mintChild(address,uint256,string)'))
    */
    function mintChildFor(address to, uint256 tokenId, string calldata label,
bytes calldata signature) external {
        _validate(
            keccak256(abi.encodeWithSelector(0xd8d3cc6e, to, tokenId,
label)),
            tokenId,
            signature
        );
        _registry.controlledMintChild(to, tokenId, label);
    }

    /*

```

```

        * 0xce9fb82b ==
bytes4(keccak256('safeMintChild(address,uint256,string,bytes)'))
    */
    function safeMintChildFor(address to, uint256 tokenId, string calldata
label, bytes calldata _data, bytes calldata signature) external {
        _validate(
            keccak256(abi.encodeWithSelector(0xce9fb82b, to, tokenId, label,
_data)),
            tokenId,
            signature
        );
        _registry.controlledSafeMintChild(to, tokenId, label, _data);
    }

    /*
    * 0x7c69eae2 ==
bytes4(keccak256('safeMintChild(address,uint256,string)'))
    */
    function safeMintChildFor(address to, uint256 tokenId, string calldata
label, bytes calldata signature) external {
        _validate(
            keccak256(abi.encodeWithSelector(0x7c69eae2, to, tokenId,
label)),
            tokenId,
            signature
        );
        _registry.controlledSafeMintChild(to, tokenId, label, "");
    }

    /*
    * 0x9e5be9a5 ==
bytes4(keccak256('transferFromChild(address,address,uint256,string)'))
    */
    function transferFromChildFor(
        address from,
        address to,
        uint256 tokenId,
        string calldata label,
        bytes calldata signature
    )
    external
    {
        _validate(
            keccak256(abi.encodeWithSelector(0x9e5be9a5, from, to, tokenId,
label)),
            tokenId,
            signature
        );
        _registry.controlledTransferFrom(from, to,
_registry.childIdOf(tokenId, label));
    }

    /*
    * 0xc29b52f9 ==
bytes4(keccak256('safeTransferFromChild(address,address,uint256,string,bytes)
'))
    */

```

```

function safeTransferFromChildFor(
    address from,
    address to,
    uint256 tokenId,
    string calldata label,
    bytes calldata _data,
    bytes calldata signature
)
    external
{
    _validate(
        keccak256(abi.encodeWithSelector(0xc29b52f9, from, to, tokenId,
label, _data)),
        tokenId,
        signature
    );
    _registry.controlledSafeTransferFrom(from, to,
_registry.childIdOf(tokenId, label), _data);
}

/*
 * 0x9d743989 ==
bytes4(keccak256('safeTransferFromChild(address,address,uint256,string)'))
 */
function safeTransferFromChildFor(
    address from,
    address to,
    uint256 tokenId,
    string calldata label,
    bytes calldata signature
)
    external
{
    _validate(
        keccak256(abi.encodeWithSelector(0x9d743989, from, to, tokenId,
label)),
        tokenId,
        signature
    );
    _registry.controlledSafeTransferFrom(from, to,
_registry.childIdOf(tokenId, label), "");
}

/*
 * 0x5cbell112 == bytes4(keccak256('burnChild(uint256,string)'))
 */
function burnChildFor(uint256 tokenId, string calldata label, bytes
calldata signature) external {
    _validate(
        keccak256(abi.encodeWithSelector(0x5cbell112, tokenId, label)),
        tokenId,
        signature
    );
    _registry.controlledBurn(_registry.childIdOf(tokenId, label));
}

/*

```

```

        * 0x2392c189 == bytes4(keccak256('resolveTo(address,uint256)'))
        */
        function resolveToFor(address to, uint256 tokenId, bytes calldata
signature) external {
            _validate(
                keccak256(abi.encodeWithSelector(0x2392c189, to, tokenId)),
                tokenId,
                signature
            );
            _registry.controlledResolveTo(to, tokenId);
        }
    }
}

```

## IURIPrefixController.sol

```

pragma solidity 0.5.12;

interface IURIPrefixController {

    /**
     * @dev Function to control the token URI metadata prefix (base url).
     * @param prefix string to set the base url to.
     */
    function setTokenURIPrefix(string calldata prefix) external;
}

```

## URIPrefixController.sol

```

pragma solidity 0.5.12;

import "@openzeppelin/contracts/access/roles/WhitelistedRole.sol";
import "../IURIPrefixController.sol";
import "../Registry.sol";

/**
 * @title URIPrefixController
 * @dev Defines the functions for distribution of Second Level Domains
(SLD)s.
 */
contract URIPrefixController is IURIPrefixController, WhitelistedRole {

    Registry internal _registry;

    constructor (Registry registry) public {
        _registry = registry;
    }

    function registry() external view returns (address) {
        return address(_registry);
    }
}

```

```
function setTokenURIPrefix(string calldata prefix) external  
onlyWhitelisted {  
    _registry.controlledSetTokenURIPrefix(prefix);  
}  
  
}
```

# Smart Contract Architecture & Administration

## Legend

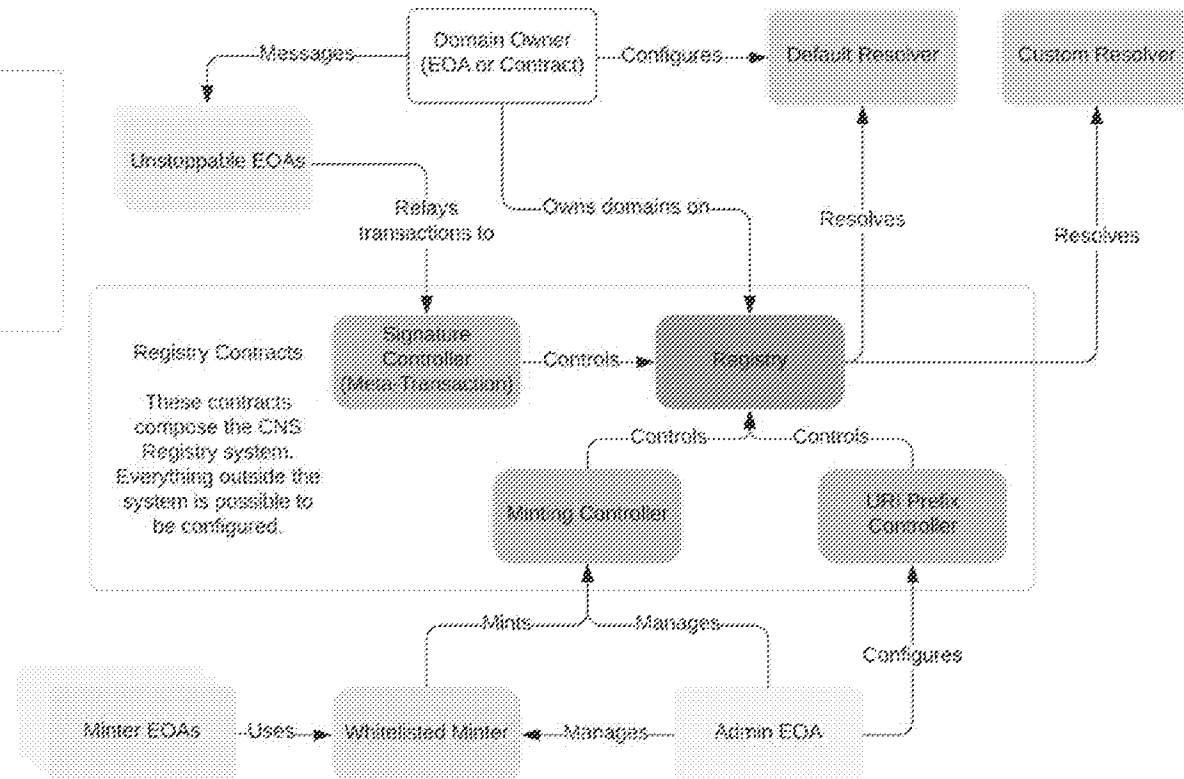
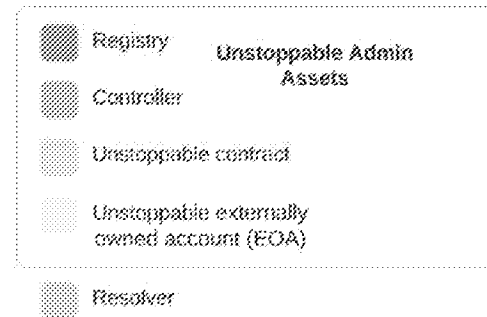


FIG. 1

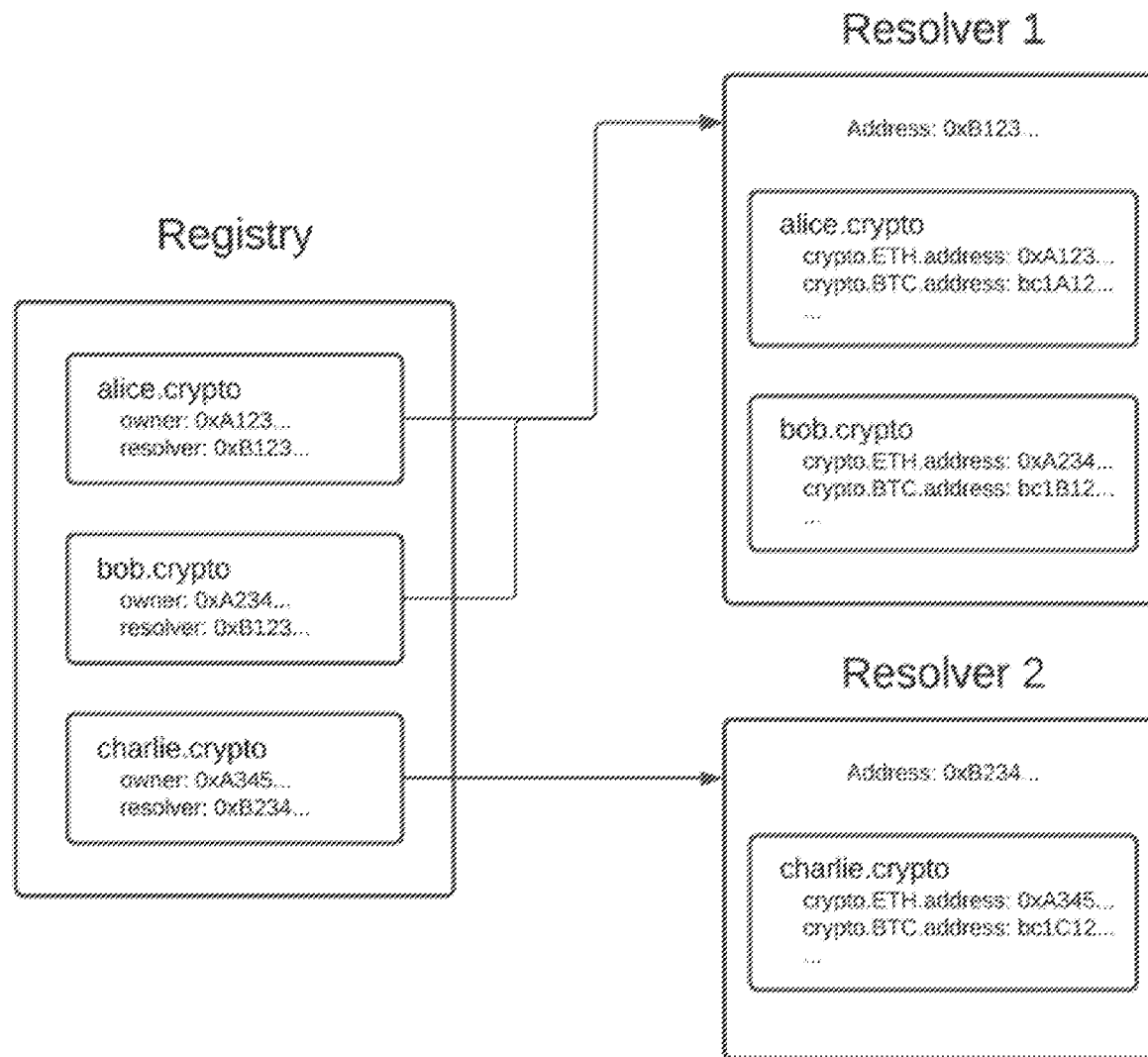


FIG. 2

## Provide domain records via Proxy Reader Smart Contract

### Legend

**Registry Smart Contract** - Domains registry which stores information about domains ownership and resolvers

**Resolver Smart Contract** - Domain records storage. There are multiple resolvers exists in Ethereum network. Domain can be linked to one Resolver at the same time.

**Proxy Reader Smart Contract** - Helper for decreasing amount of HTTP calls to Ethereum node.

[HTTP] - Call via network protocols

[EVM] - Local Smart Contract function call

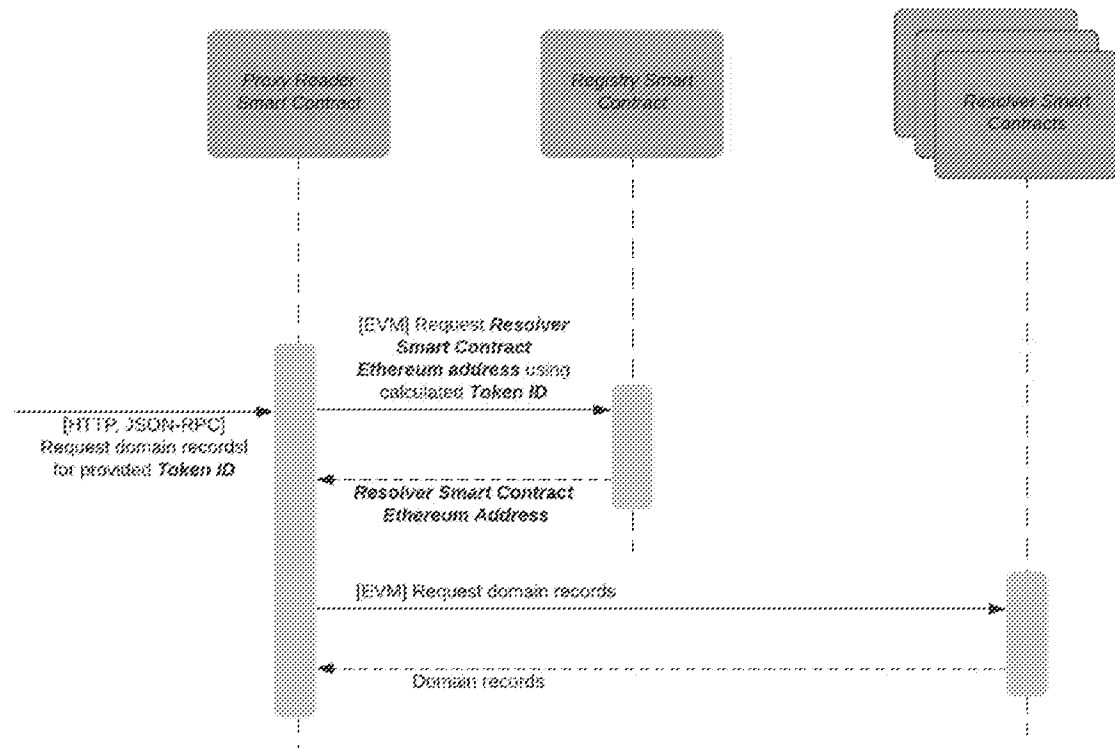


FIG. 3

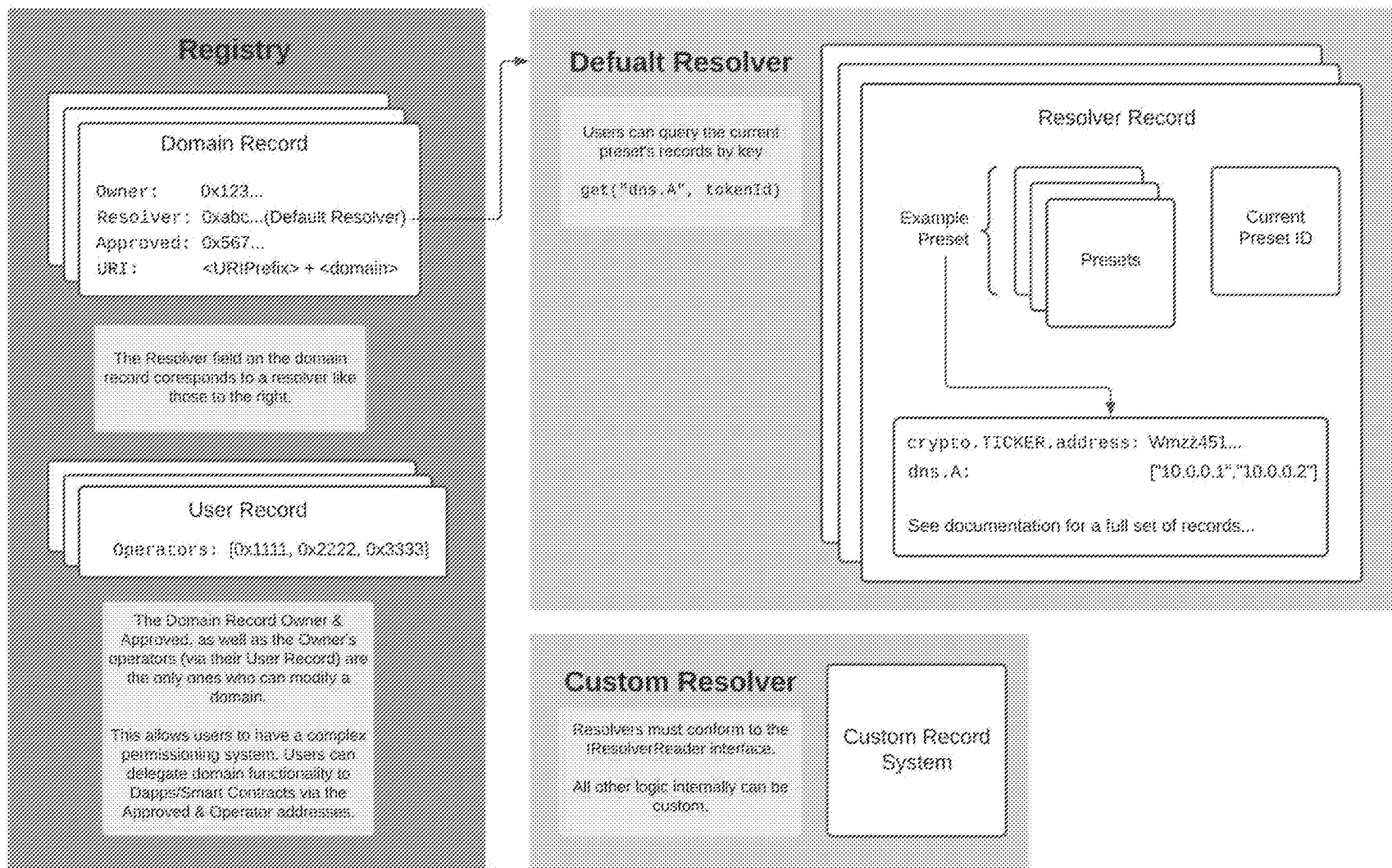


FIG. 4

## Example Meta-Transaction Flow

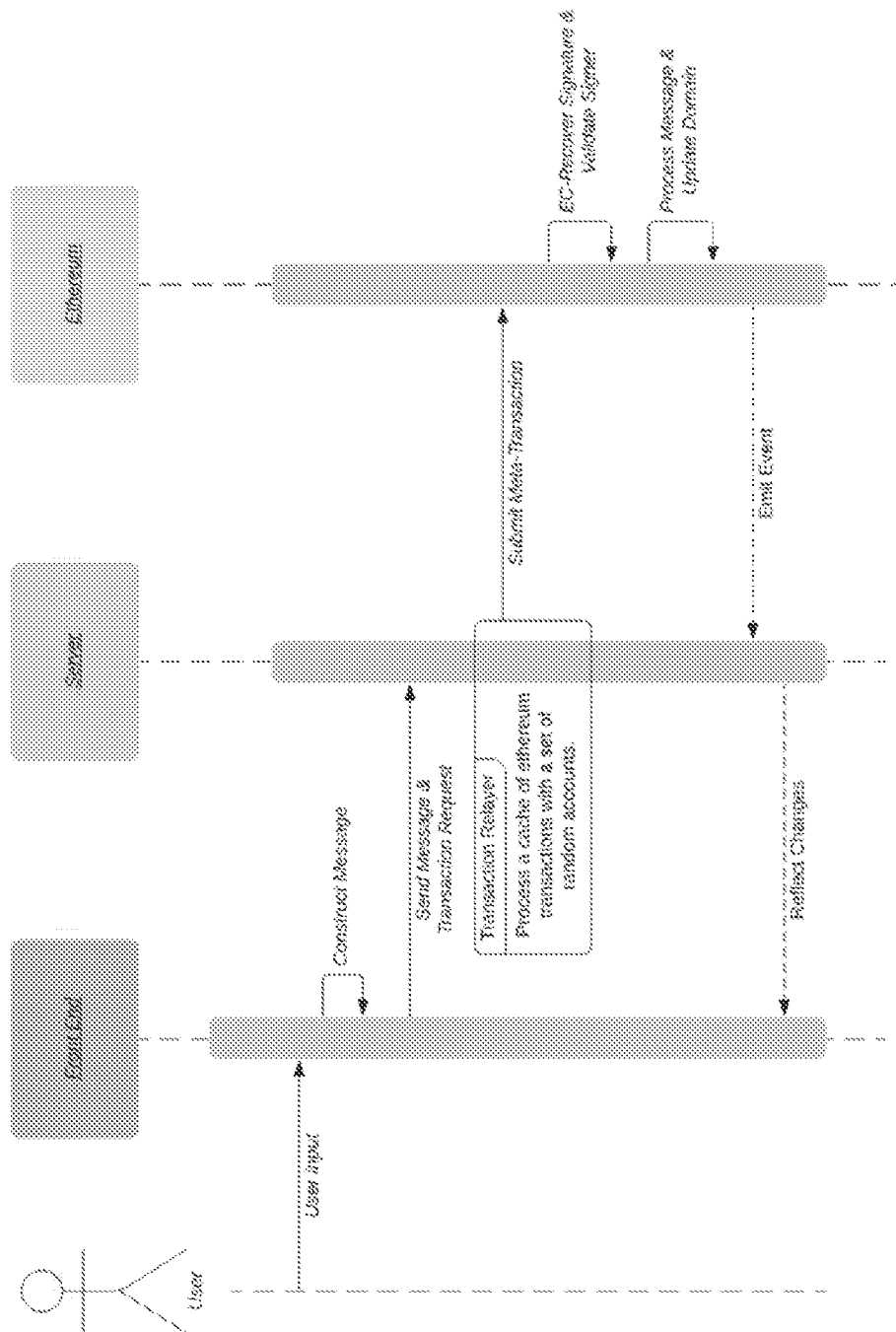
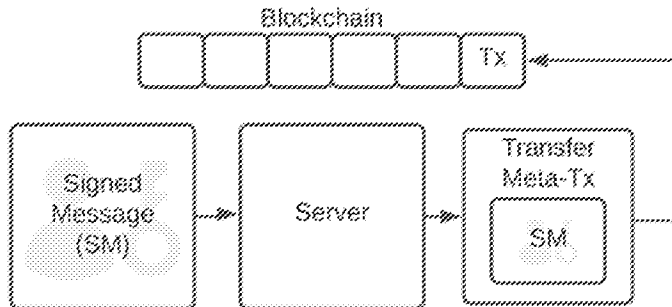


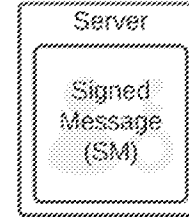
FIG. 5

## Without Nonces

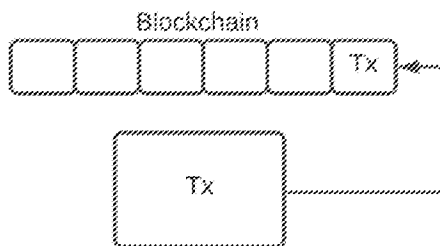
1. Submits transaction to blockchain using a meta-transaction



2. Server keeps SM for later.



3. Another transaction is later submitted to blockchain.



4. Server submits same meta-transaction, replaying state changes.

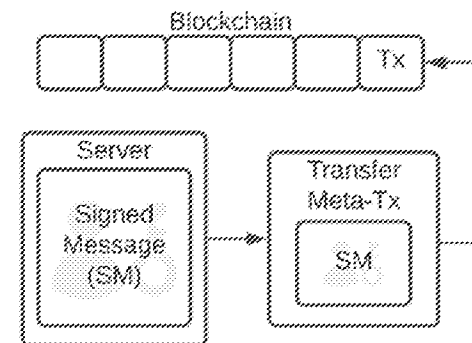


FIG. 6

## With Nonces

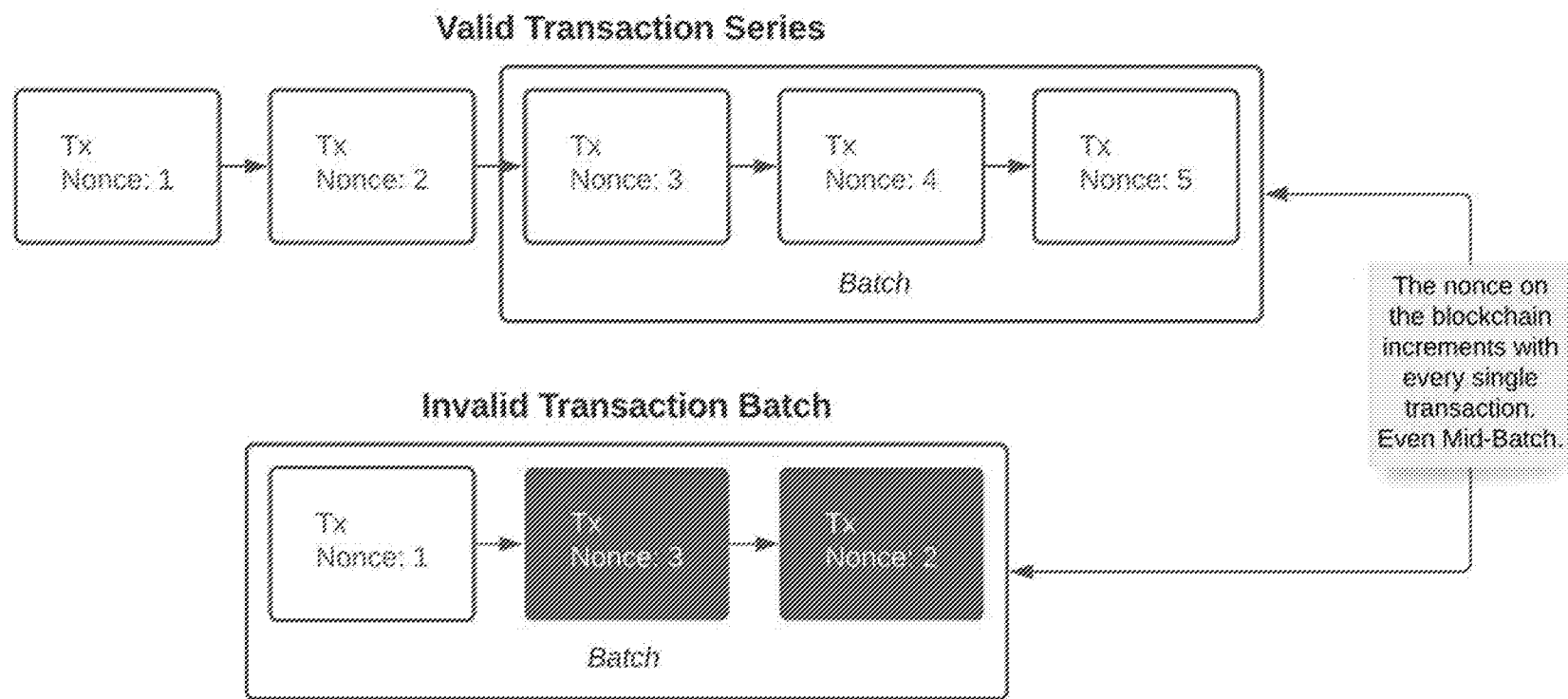


FIG. 7

## Overview: Read DWeb website from Ethereum and Decentralized network

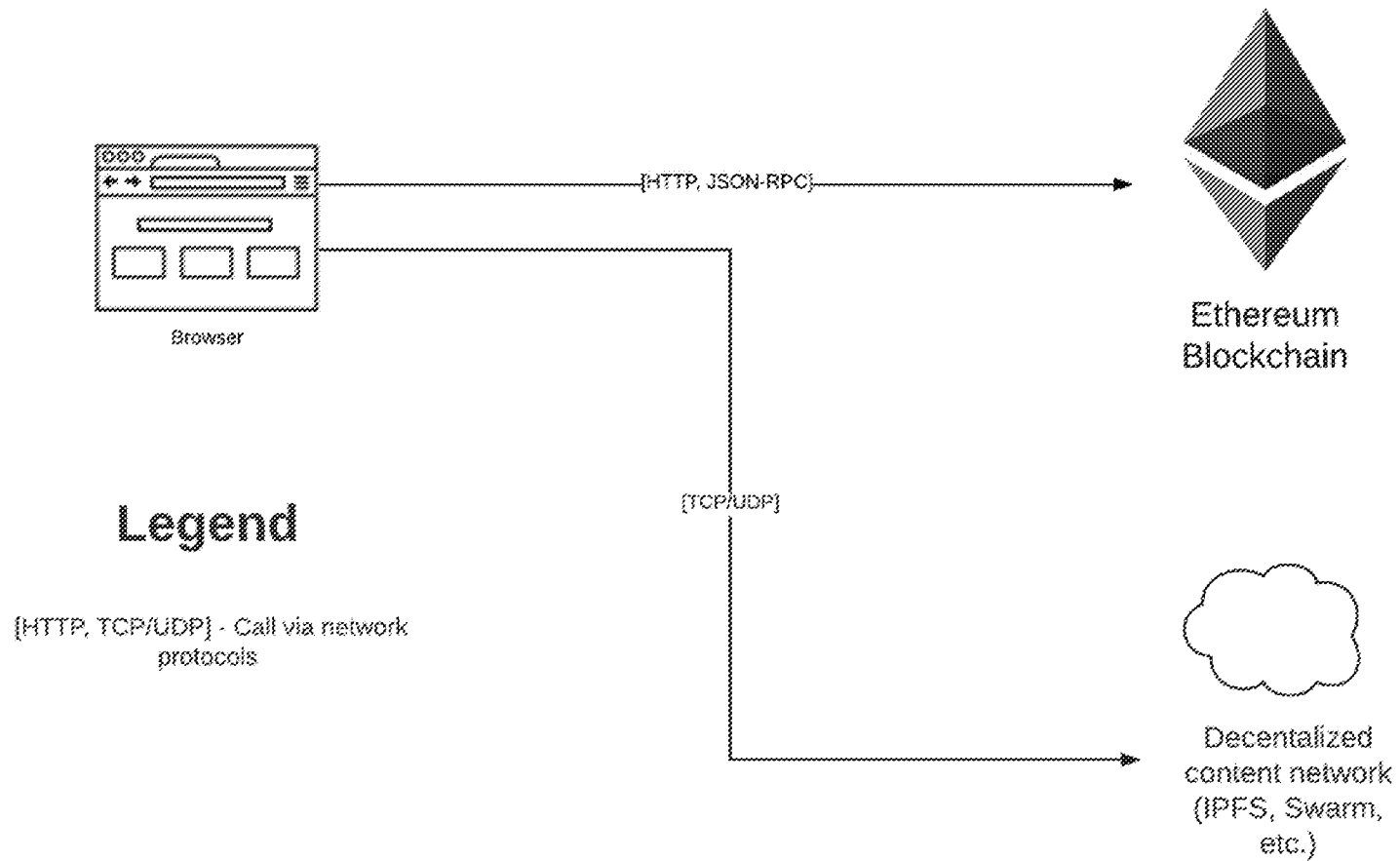


FIG. 8

## Configure DNS gateway

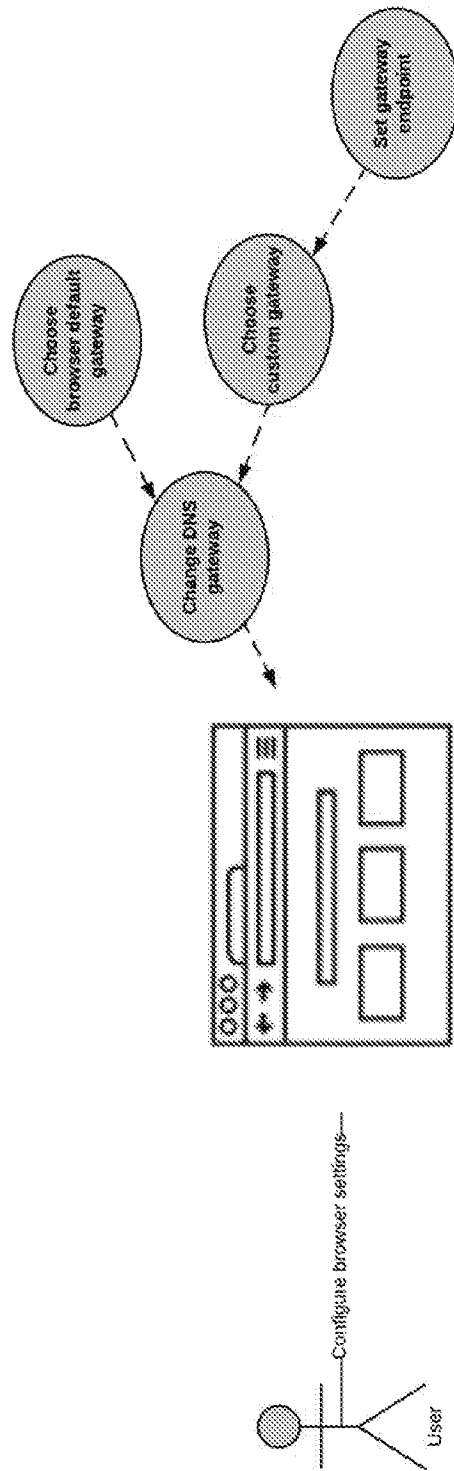


FIG. 9

## Overview: DWeb website via DNS + DWeb gateways

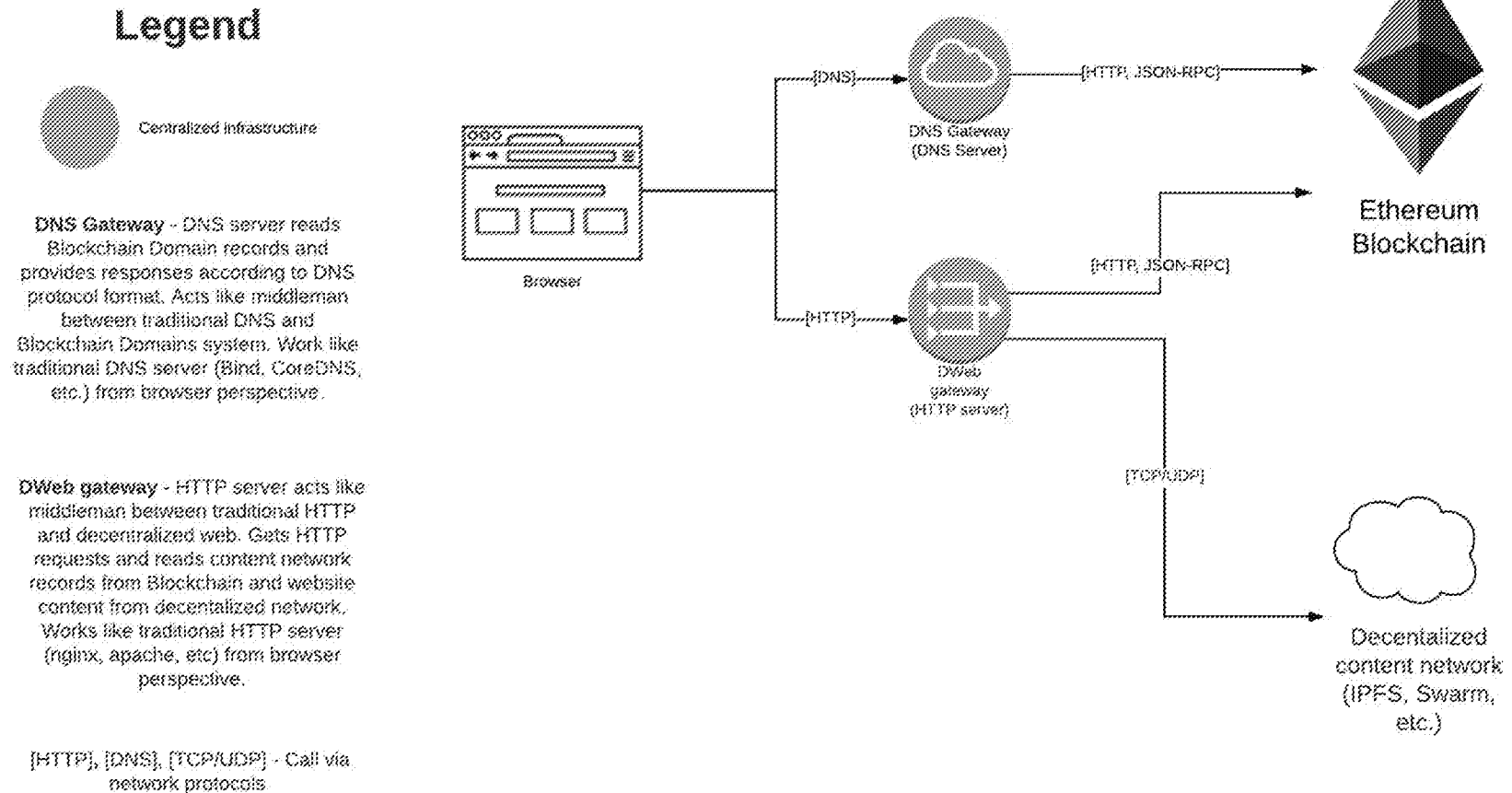
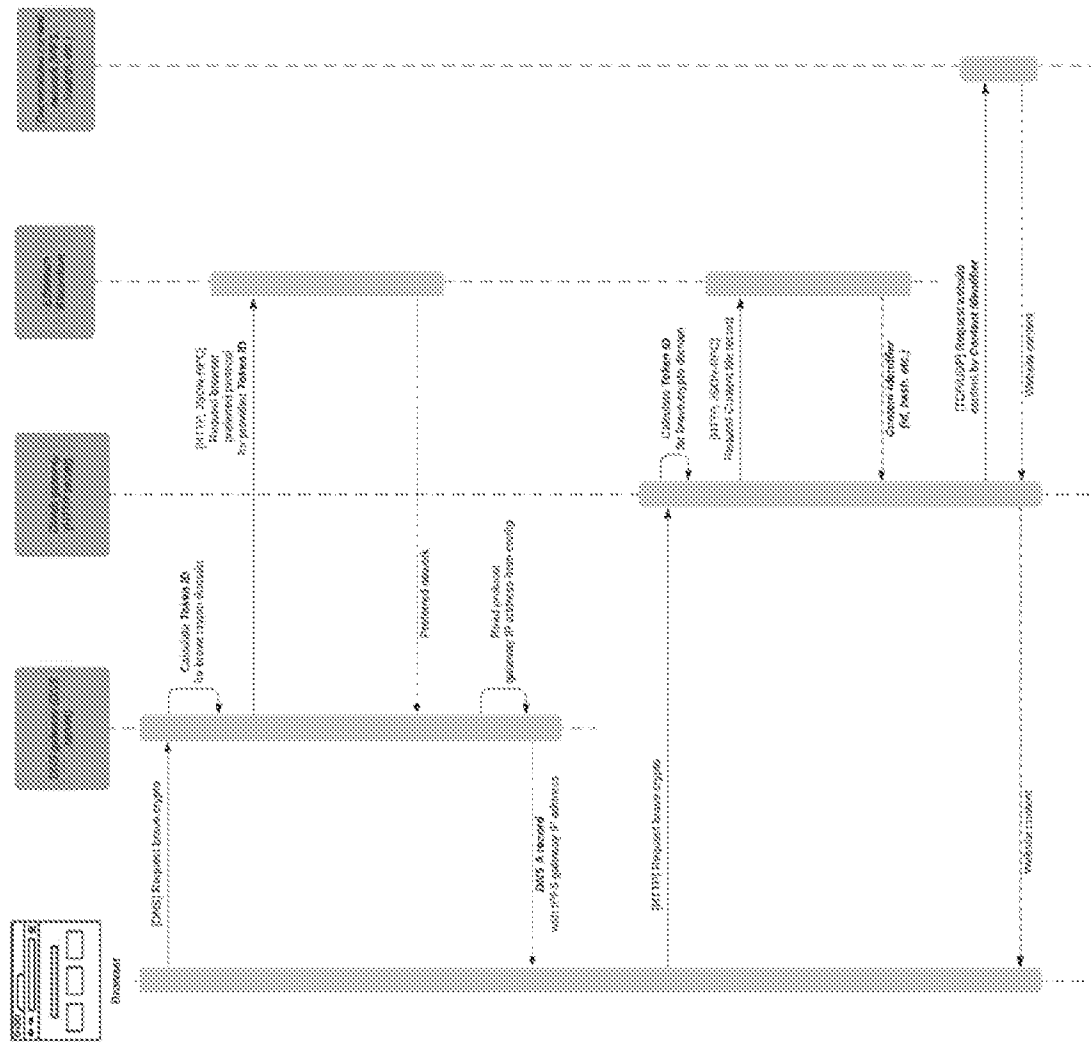


FIG. 10

# Resolve DWeb website via DNS gateway and DWeb gateway



## Legend

**Token ID** - Domain unique identifier which can be calculated from domain name using *namehash algorithm*

**DNS Gateway** - DNS server sends Blockchain Domain records and provides responses according to DNS protocol format. Acts like middleman between traditional DNS and Blockchain Domains system. Works like traditional DNS server (Bind, CloudFlare, etc.) from *user's perspective*.

**DWeb Gateway** - HTTP server acts like middleman between traditional HTTP and decentralized web. Acts HTTP requests and sends content network records from Blockchain and website content from decentralized network. Works like traditional HTTP server (nginx, apache, etc.) from *browser perspective*.

**[HTTP, 200 OK]** - Can be network protocols

FIG. 11

## Resolve DWeb website by direct reading from Ethereum and Decentralized network

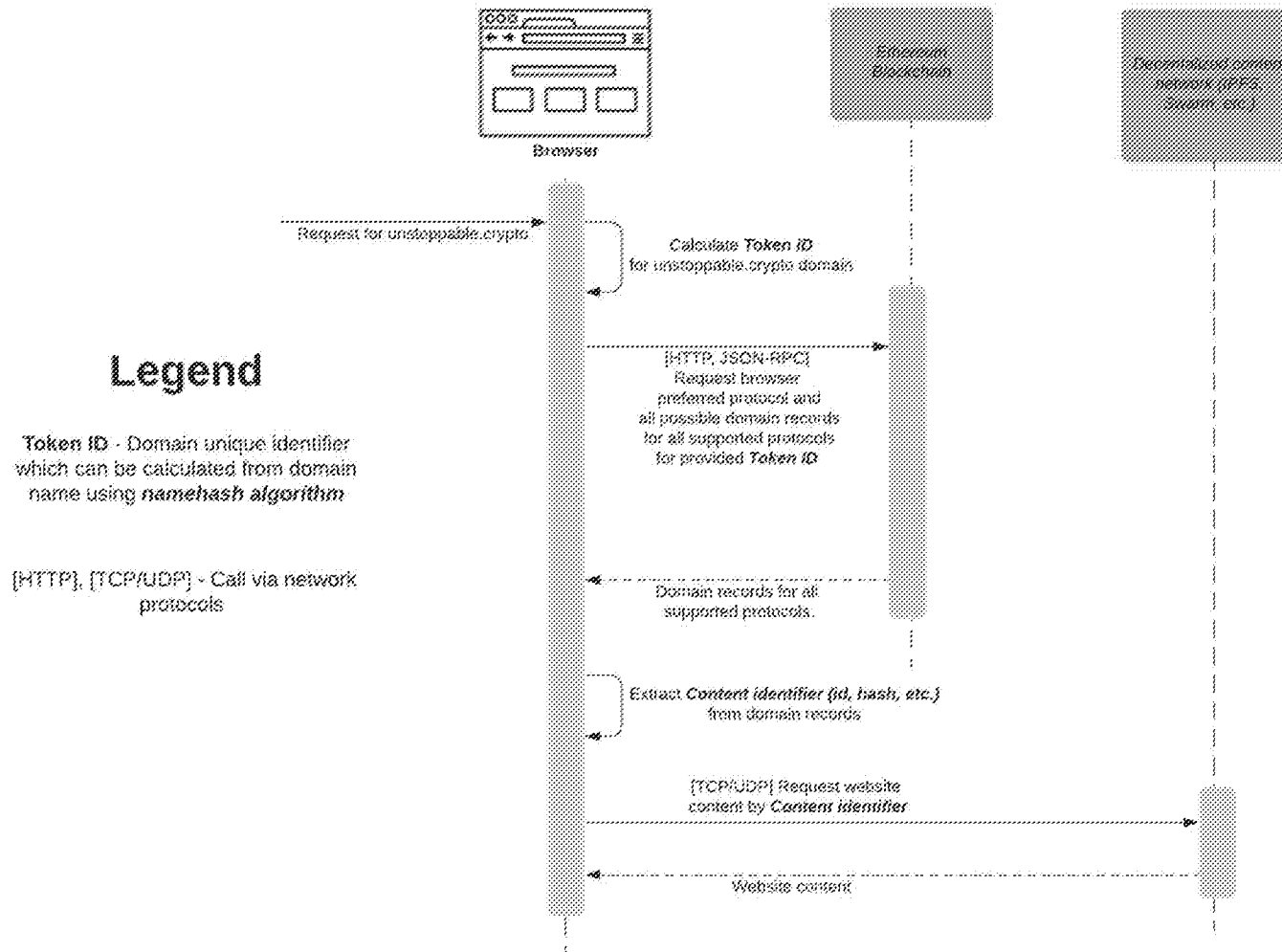


FIG. 12

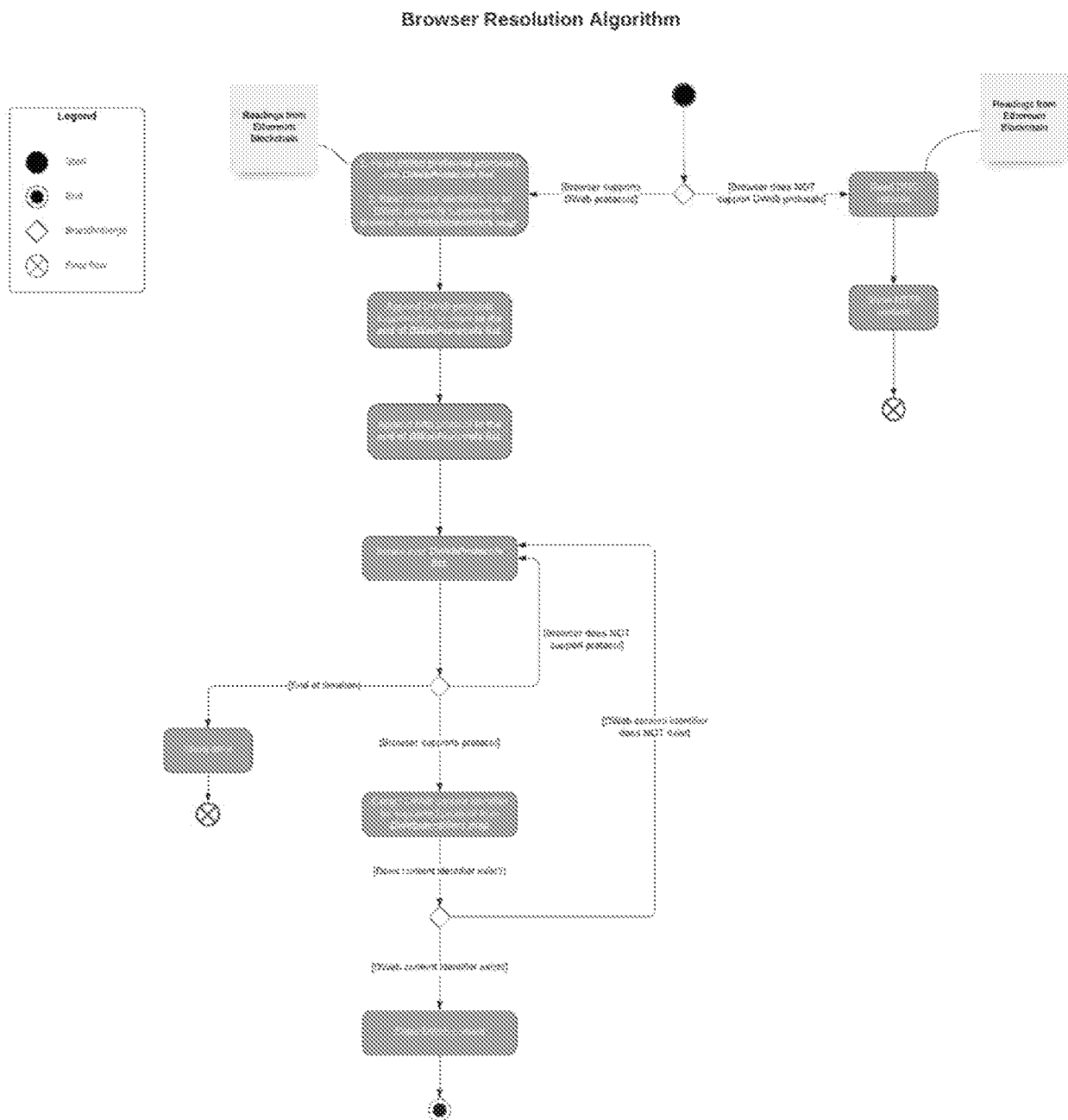


**FIG. 13**