INTERNET ARCHIVE
archive.org

AFFIDAVIT OF NATHANIEL E FRANK-WHITE

1. I am a Records Request Processor at the Internet Archive. I make this declaration of my own personal knowledge.

2. The Internet Archive is a website that provides access to a digital library of Internet sites and other cultural artifacts in digital form. Like a paper library, we provide free access to researchers, historians, scholars, and the general public. The Internet Archive has partnered with and receives support from various institutions, including the Library of Congress.

3. The Internet Archive has created a service known as the Wayback Machine. The Wayback Machine makes it possible to browse more than 450 billion pages stored in the Internet Archive's web archive. Visitors to the Wayback Machine can search archives by URL (i.e., a website address). If archived records for a URL are available, the visitor will be presented with a display of available dates. The visitor may select one of those dates, and begin browsing an archived version of the Web. Links on archived files in the Wayback Machine point to other archived files (whether HTML pages or other file types), if any are found for the URL indicated by a given link. For instance, the Wayback Machine is designed such that when a visitor clicks on a hyperlink on an archived page that points to another URL, the visitor will be served the archived file found for the hyperlink's URL with the closest available date to the initial file containing the hyperlink.

4. The archived data made viewable and browsable by the Wayback Machine is obtained by use of web archiving software that automatically stores copies of files available via the Internet, each file preserved as it existed at a particular point in time.

5. The Internet Archive assigns a URL on its site to the archived files in the format http://web.archive.org/web/[Year in yyyy][Month in mm][Day in dd][Time code in hh:mm:ss]/[Archived URL] aka an "extended URL". Thus, the extended URL http://web.archive.org/web/19970126045828/http://www.archive.org/ would be the URL for the record of the Internet Archive home page HTML file (http://www.archive.org/) archived on January 26, 1997 at 4:58 a.m. and 28 seconds (1997/01/26 at 04:58:28). The date indicated by an extended URL applies to a preserved instance of a file for a given URL, but not necessarily to any other files linked therein. Thus, in the case of a page constituted by a primary HTML file and other separate files (e.g., files with images, audio, multimedia, design elements, or other embedded content) linked within that primary HTML file, the primary HTML file and the other files will each have their own respective extended URLs and may not have been archived on the same dates.

6. Attached hereto as Exhibit A are true and accurate copies of browser screenshots of the Internet Archive's records of the archived files for the URLs and the dates specified in the attached coversheet of each printout.

INTERNET ARCHIVE
archive.org

7. Attached hereto as Exhibit B are true and accurate copies of the Internet Archive's records of the archived files for the URLs and the dates specified in the attached coversheet of each file.

8. I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

DATE: 01/27/2024

Nathaniel Frank-White
Nathaniel E Frank-White

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

EXHIBIT A

https://web.archive.org/web/20180329093822/https://eips.ethereum.org/EIPS/eip-137

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

INTERNET ARCHIVE Wayback Machine https://eips.ethereum.org/EIPS/eip-137 Go FEB MAR OCT 29 2017 2018 2019 About this capture
76 captures

EIP 137: Ethereum Domain Name Service - Specification ‹›

Author	Nick Johnson
Status	Final
Type	Standards Track
Category	ERC
Created	2016-04-04

Abstract

This draft EIP describes the details of the Ethereum Name Service, a proposed protocol and ABI definition that provides flexible resolution of short, human-readable names to service and resource identifiers. This permits users and developers to refer to human-readable and easy to remember names, and permits those names to be updated as necessary when the underlying resource (contract, content-addressed data, etc) changes.

The goal of domain names is to provide stable, human-readable identifiers that can be used to specify network resources. In this way, users can enter a memorable string, such as 'vitalik.wallet' or 'www.mysite.swarm', and be directed to the appropriate resource. The mapping between names and resources may change over time, so a user may change wallets, a website may change hosts, or a swarm document may be updated to a new version, without the domain name changing. Further, a domain need not specify a single resource; different record types allow the same domain to reference different resources. For instance, a browser may resolve 'mysite.swarm' to the IP address of its server by fetching its A (address) record, while a mail client may resolve the same address to a mail server by fetching its MX (mail exchanger) record.

Motivation

Existing specifications and implementations for name resolution in Ethereum provide basic functionality, but suffer several shortcomings that will significantly limit their long-term usefulness:

- A single global namespace for all names with a single 'centralised' resolver.
- Limited or no support for delegation and sub-names/sub-domains.
- Only one record type, and no support for associating multiple copies of a record with a domain.
- Due to a single global implementation, no support for multiple different name allocation systems.
- Conflation of responsibilities: Name resolution, registration, and whois information.

Use-cases that these features would permit include:

- Support for subnames/sub-domains - eg, live.mysite.tld and forum.mysite.tld.
- Multiple services under a single name, such as a DApp hosted in Swarm, a Whisper address, and a mail server.
- Support for DNS record types, allowing blockchain hosting of 'legacy' names. This would permit an Ethereum client such as Mist to resolve the address of a traditional website, or the mail server for an email address, from a blockchain name.
- DNS gateways, exposing ENS domains via the Domain Name Service, providing easier means for legacy clients to resolve and connect to blockchain services.

The first two use-cases, in particular, can be observed everywhere on the present-day internet under DNS, and we believe them to be fundamental features of a name service that will continue to be useful

as the Ethereum platform develops and matures.

The normative parts of this document does not specify an implementation of the proposed system; its purpose is to document a protocol that different resolver implementations can adhere to in order to facilitate consistent name resolution. An appendix provides sample implementations of resolver contracts and libraries, which should be treated as illustrative examples only.

Likewise, this document does not attempt to specify how domains should be registered or updated, or how systems can find the owner responsible for a given domain. Registration is the responsibility of registrars, and is a governance matter that will necessarily vary between top-level domains.

Updating of domain records can also be handled separately from resolution. Some systems, such as swarm, may require a well defined interface for updating domains, in which event we anticipate the development of a standard for this.

Specification

Overview

The ENS system comprises three main parts:

- The ENS registry
- Resolvers
- Registrars

The registry is a single contract that provides a mapping from any registered name to the resolver responsible for it, and permits the owner of a name to set the resolver address, and to create subdomains, potentially with different owners to the parent domain.

Resolvers are responsible for performing resource lookups for a name - for instance, returning a contract address, a content hash, or IP address(es) as appropriate. The resolver specification, defined here and extended in other EIPs, defines what methods a resolver may implement to support resolving different types of records.

Registrars are responsible for allocating domain names to users of the system, and are the only entities capable of updating the ENS; the owner of a node in the ENS registry is its registrar. Registrars may be contracts or externally owned accounts, though it is expected that the root and top-level registrars, at a minimum, will be implemented as contracts.

Resolving a name in ENS is a two-step process. First, the ENS registry is called with the name to resolve, after hashing it using the procedure described below. If the record exists, the registry returns the address of its resolver. Then, the resolver is called, using the method appropriate to the resource being requested. The resolver then returns the desired result.

For example, suppose you wish to find the address of the token contract associated with 'beercoin.eth'. First, get the resolver:

```
var node = namehash("beercoin.eth");
var resolver = ens.resolver(node);
```

Then, ask the resolver for the address for the contract:

```
var hash = resolver.addr(node);
```

Because the `namehash` procedure depends only on the name itself, this can be precomputed and inserted into a contract, removing the need for string manipulation, and permitting O(1) lookup of ENS records regardless of the number of components in the raw name.

Name Syntax

ENS names must conform to the following syntax:

```
<domain> ::= <label> | <domain> "." <label>
<label> ::= any valid string label per [UTS46](http://unicode.org/reports/tr46/)
```

In short, names consist of a series of dot-separated labels. Each label must be a valid normalised label as described in UTS46 with the options `transitional=false` and `useSTD3AsciiRules=true`. For Javascript implementations, a library is available that normalises and checks names.

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

Note that while upper and lower case letters are allowed in names, the UTS46 normalisation process case-folds labels before hashing them, so two names with different case but identical spelling will produce the same namehash.

Labels and domains may be of any length, but for compatibility with legacy DNS, it is recommended that labels be restricted to no more than 64 characters each, and complete ENS names to no more than 255 characters. For the same reason, it is recommended that labels do not start or end with hyphens, or start with digits.

namehash algorithm

Before being used in ENS, names are hashed using the 'namehash' algorithm. This algorithm recursively hashes components of the name, producing a unique, fixed-length string for any valid input domain. The output of namehash is referred to as a 'node'.

Pseudocode for the namehash algorithm is as follows:

```
def namehash(name):
  if name == '':
    return '\0' * 32
  else:
    label, _, remainder = name.partition('.')
    return sha3(namehash(remainder) + sha3(label))
```

Informally, the name is split into labels, each label is hashed. Then, starting with the last component, the previous output is concatenated with the label hash and hashed again. The first component is concatenated with 32 '0' bytes. Thus, 'mysite.swarm' is processed as follows:

```
node = '\0' * 32
node = sha3(node + sha3('swarm'))
node = sha3(node + sha3('mysite'))
```

Implementations should conform to the following test vectors for namehash:

```
namehash('') = 0x0000000000000000000000000000000000000000000000000000000000000000
namehash('eth') = 0x93cdeb708b7545dc668eb9280176169d1c33cfd8ed6f04690a0bcc88a93fc4ae
namehash('foo.eth') = 0xde9b09fd7c5f901e23a3f19fecc54828e9c848539801e86591bd9801b019f84f
```

Registry specification

The ENS registry contract exposes the following functions:

```
function owner(bytes32 node) constant returns (address);
```

Returns the owner (registrar) of the specified node.

```
function resolver(bytes32 node) constant returns (address);
```

Returns the resolver for the specified node.

```
function ttl(bytes32 node) constant returns (uint64);
```

Returns the time-to-live (TTL) of the node; that is, the maximum duration for which a node's information may be cached.

```
function setOwner(bytes32 node, address owner);
```

Transfers ownership of a node to another registrar. This function may only be called by the current owner of `node`. A successful call to this function logs the event `Transfer(bytes32 indexed, address)`.

```
function setSubnodeOwner(bytes32 node, bytes32 label, address owner);
```

Creates a new node, `sha3(node, label)` and sets its owner to `owner`, or updates the node with a new owner if it already exists. This function may only be called by the current owner of `node`. A successful call to this function logs the event `NewOwner(bytes32 indexed, bytes32 indexed, address)`.

```
function setResolver(bytes32 node, address resolver);
```

Sets the resolver address for `node`. This function may only be called by the owner of `node`. A successful call to this function logs the event `NewResolver(bytes32 indexed, address)`.

```
function setTTL(bytes32 node, uint64 ttl);
```

Sets the TTL for a node. A node's TTL applies to the 'owner' and 'resolver' records in the registry, as well as to any information returned by the associated resolver.

Resolver specification

Resolvers may implement any subset of the record types specified here. Where a record types specification requires a resolver to provide multiple functions, the resolver MUST implement either all or none of them. Resolvers MUST specify a fallback function that throws.

Resolvers have one mandatory function:

```
function supportsInterface(bytes4 interfaceID) constant returns (bool)
```

The `supportsInterface` function is documented in EIP 165, and returns true if the resolver implements the interface specified by the provided 4 byte identifier. An interface identifier consists of the XOR of the function signature hashes of the functions provided by that interface; in the degenerate case of single-function interfaces, it is simply equal to the signature hash of that function. If a resolver returns `true` for `supportsInterface()`, it must implement the functions specified in that interface.

`supportsInterface` must always return true for `0x01ffc9a7`, which is the interface ID of `supportsInterface` itself.

Currently standardised resolver interfaces are specified in the table below.

The following interfaces are defined:

Interface name	Interface hash	Specification
`addr`	0x3b3b57de	Contract address
`name`	0x691f3431	#181
`ABI`	0x2203ab56	#205
`pubkey`	0xc8690233	#619

EIPs may define new interfaces to be added to this registry.

Contract Address Interface

Resolvers wishing to support contract address resources must provide the following function:

```
function addr(bytes32 node) constant returns (address);
```

If the resolver supports `addr` lookups but the requested node does not have a record, the resolver MUST return the zero address.

Clients resolving the `addr` record MUST check for a zero return value, and treat this in the same manner as a name that does not have a resolver specified - that is, refuse to send funds to or interact with the address. Failure to do this can result in users accidentally sending funds to the 0 address.

Changes to an address MUST trigger the following event:

```
event AddrChanged(bytes32 indexed node, address a);
```

Appendix A: Registry Implementation

```
contract ENS {
    struct Record {
```

```
    struct Record {
        address owner;
        address resolver;
        uint64 ttl;
    }

    mapping(bytes32=>Record) records;

    event NewOwner(bytes32 indexed node, bytes32 indexed label, address owner);
    event Transfer(bytes32 indexed node, address owner);
    event NewResolver(bytes32 indexed node, address resolver);

    modifier only_owner(bytes32 node) {
        if(records[node].owner != msg.sender) throw;
        _
    }

    function ENS(address owner) {
        records[0].owner = owner;
    }

    function owner(bytes32 node) constant returns (address) {
        return records[node].owner;
    }

    function resolver(bytes32 node) constant returns (address) {
        return records[node].resolver;
    }

    function ttl(bytes32 node) constant returns (uint64) {
        return records[node].ttl;
    }

    function setOwner(bytes32 node, address owner) only_owner(node) {
        Transfer(node, owner);
        records[node].owner = owner;
    }

    function setSubnodeOwner(bytes32 node, bytes32 label, address owner) only_owner(node
        var subnode = sha3(node, label);
        NewOwner(node, label, owner);
        records[subnode].owner = owner;
    }

    function setResolver(bytes32 node, address resolver) only_owner(node) {
        NewResolver(node, resolver);
        records[node].resolver = resolver;
    }

    function setTTL(bytes32 node, uint64 ttl) only_owner(node) {
        NewTTL(node, ttl);
        records[node].ttl = ttl;
    }
}
```

Appendix B: Sample Resolver Implementations

Built-in resolver

The simplest possible resolver is a contract that acts as its own name resolver by implementing the contract address resource profile:

```
contract DoSomethingUseful {
    // Other code

    function addr(bytes32 node) constant returns (address) {
        return this;
    }
```


Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

```
    function supportsInterface(bytes4 interfaceID) constant returns (bool) {
        return interfaceID == 0x3b3b57de || interfaceID == 0x01ffc9a7;
    }

    function() {
        throw;
    }
}
```

Such a contract can be inserted directly into the ENS registry, eliminating the need for a separate resolver contract in simple use-cases. However, the requirement to 'throw' on unknown function calls may interfere with normal operation of some types of contract.

Standalone resolver

A basic resolver that implements the contract address profile, and allows only its owner to update records:

```
contract Resolver {
    event AddrChanged(bytes32 indexed node, address a);

    address owner;
    mapping(bytes32=>address) addresses;

    modifier only_owner() {
        if(msg.sender != owner) throw;
        _
    }

    function Resolver() {
        owner = msg.sender;
    }

    function addr(bytes32 node) constant returns(address) {
        return addresses[node];
    }

    function setAddr(bytes32 node, address addr) only_owner {
        addresses[node] = addr;
        AddrChanged(node, addr);
    }

    function supportsInterface(bytes4 interfaceID) constant returns (bool) {
        return interfaceID == 0x3b3b57de || interfaceID == 0x01ffc9a7;
    }

    function() {
        throw;
    }
}
```

After deploying this contract, use it by updating the ENS registry to reference this contract for a name, then calling `setAddr()` with the same node to set the contract address it will resolve to.

Public resolver

Similar to the resolver above, this contract only supports the contract address profile, but uses the ENS registry to determine who should be allowed to update entries:

```
contract PublicResolver {
    event AddrChanged(bytes32 indexed node, address a);
    event ContentChanged(bytes32 indexed node, bytes32 hash);

    ENS ens;
    mapping(bytes32=>address) addresses;

    modifier only_owner(bytes32 node) {
        if(ens.owner(node) != msg.sender) throw;
```


Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

```
        _
    }

    function PublicResolver(address ensAddr) {
        ens = ENS(ensAddr);
    }

    function addr(bytes32 node) constant returns (address ret) {
        ret = addresses[node];
    }

    function setAddr(bytes32 node, address addr) only_owner(node) {
        addresses[node] = addr;
        AddrChanged(node, addr);
    }

    function supportsInterface(bytes4 interfaceID) constant returns (bool) {
        return interfaceID == 0x3b3b57de || interfaceID == 0x01ffc9a7;
    }

    function() {
        throw;
    }
}
```

Appendix C: Sample Registrar Implementation

This registrar allows users to register names at no cost if they are the first to request them.

```
contract FIFSRegistrar {
    ENS ens;
    bytes32 rootNode;

    function FIFSRegistrar(address ensAddr, bytes32 node) {
        ens = ENS(ensAddr);
        rootNode = node;
    }

    function register(bytes32 subnode, address owner) {
        var node = sha3(rootNode, subnode);
        var currentOwner = ens.owner(node);
        if(currentOwner != 0 && currentOwner != msg.sender)
            throw;

        ens.setSubnodeOwner(rootNode, subnode, owner);
    }
}
```

Ethereum Improvement Proposals

Ethereum Improvement Proposals ethereum/EIPs ethereum

Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards.

https://web.archive.org/web/20190815193536/https://medium.com/the-ethereum-name-service/ens-now-supports-tor-onion-address-resolution-9bb3bdff6217

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

INTERNET ARCHIVE Wayback Machine https://medium.com/the-ethereum-name-service/ens-now-supports-tor-onion-address-resolution-9bb3bdff8217 Go JUL AUG JAN 7 captures 15 2018 2019 2020 About this capture

ENS Now Supports Tor .Onion Address Resolution



Brantly Millegan Follow
Aug 13 · 4 min read

We are pleased to announce that the Ethereum Name Service (ENS) now supports the resolution of Tor .onion addresses! This is just another way ENS is supporting things beyond the Ethereum ecosystem.

The Tor network is a part of the Internet that hides the IP addresses of users. You can access it with the Tor Browser, managed by the Tor Project. You can also set up websites that are only accessible via the Tor network, called onion services.

These Tor-only web addresses end with ".onion," but otherwise the rest of the address is randomly generated. For example, to access the onion version of the privacy-centric search engine Duck Duck Go, *in your Tor Browser* (this will not work in a normal browser) you go to **https://3g2upl4pq6kufc4m.onion/**.

Trying to get human-readable .onion addresses has been an ongoing problem for many years, given the high bar for security. Since ENS is a

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

decentralized naming service that runs as a set of smart-contracts on the Ethereum blockchain, we believe ENS can be helpful tool to solve this problem.

Try It Right Now

1. **Open the Tor Browser**. If you don't already have it, you can download it here.
2. **Enable the Ethereum browser extension MetaMask.** You can get it in the Firefox add-on store (Tor Browser is Firefox-based).
3. **Enter the following example ENS name** into the address bar and hit Enter: **duckduckgotor.eth** (I set this up recently to show off the new capability)
4. **Wait a few seconds** while ENS, Metamask, and the Tor Browser do their magic, and… voilà, you should be seeing the onion version of Duck Duck Go!

How it Works

Here's what's going on to make this work:

1. After you enter duckduckgotor.eth in the address bar of the Tor Browser, MetaMask recognizes that duckduckgotor.eth is an ENS name and so prevents the browser from treating it as a search or normal DNS name.
2. MetaMask does a look-up on ENS on Ethereum, grabbing the .onion address for Duck Duck Go in the "Content" field (the same field that also accepts IPFS and Swarm hashes) in the ENS record for duckduckgotor.eth and returning it to the address bar in the Tor Browser.
3. Since what is returned is a .onion address, the Tor Browser then navigates to the onion website at that address.

How to Set Up Your Own ENS Name to Resolve to a .Onion Address

Setting up an ENS name to resolve to a .onion address in the Tor Browser is very simple.

First, you need an ENS name. You can register a .eth name with our Manager in an Ethereum-enabled browser (e.g. Tor Browser/Chrome/et al + MetaMask) and some ETH.

Second, you set up your ENS record. You can do this with our Manager in an Ethereum-enabled browser and some ETH.

Set the Resolver to the "Public Resolver" (our standard ENS record set). Then set a "Content" record that says "**onion://[insert .onion address without the .onion ending]**". Once your transactions to set this record are confirmed, you are good to go. If you already own some ETH, the whole process should only take a few minutes.

The ENS record for the example above, duckduckgotor.eth, as viewed in our Manager.

Just open up the Tor Browser with MetaMask enabled, type in your ENS name, and it should immediately be ready to resolve to the .onion address you put in the record.

Security Concerns?

If you're using Tor and an onion version of Duck Duck Go, security is likely a high priority for you. So how secure is this setup?

We will be publishing a separate blog post soon that addresses this question in greater depth than we can here. In the meantime, be careful using this setup with anything too sensitive until we're able to explain the security tradeoffs.

Who Built It

The sTorgate team at ETHNewYork

This new functionality did not come from the ENS team. It started off as a project called sTorgate at the ETHGlobal hackathon ETHNewYork in May of this year, by team members Alessandro Ricottone, Jeremy Macaluso, You Song Hou, and Luis Carbajal. Yes, this is a hackathon success story!

We were anxious to get the meat of this project into production, so Ricottone put together the necessary pull requests for the various open source projects to make this work and, after several weeks of patient waiting and prodding, they finally all got accepted.

A big thank you to the whole sTorgate team, but especially Ricottone for making this happen!

Ethereum Blockchain Cryptocurrency Tor Ethereum Name Service

201 claps



WRITTEN BY

Brantly Millegan

http://brantlymillegan.com/

Follow



The Ethereum Name Service

The official source for news related to the Ethereum Name Service (ENS). Follow this publication for the latest ENS developments.

Follow

Write the first response

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

Medium

About Help Legal

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

EXHIBIT B

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

https://web.archive.org/web/20191213174021/https://buildmedia.readthedocs.org/media/pdf/ens/latest/ens.pdf

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

ENS Documentation

Release 0.1

Nick Johnson <nick@ethereum.org>

Mar 30, 2019

ENS LABS EX1018
Page 20 of 59

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

Contents:

1 Introduction **1**
- 1.1 ENS in 5 minutes . . . 1
- 1.2 ENS on Ethereum . . . 3
- 1.3 Resources . . . 3

2 Quickstart **5**

3 User Guide **7**
- 3.1 Registering a name with the FIFS registrar . . . 7
- 3.2 Registering a name with the auction registrar . . . 8
- 3.3 Interacting with the ENS registry . . . 10

4 Implementer's Guide **13**
- 4.1 3 steps to add ENS to your application . . . 13
- 4.2 ENS libraries . . . 14
- 4.3 Resolving Names . . . 14
- 4.4 Reverse Resolution . . . 15
- 4.5 Writing a resolver . . . 15
- 4.6 Resolving names onchain . . . 16
- 4.7 Writing a registrar . . . 17
- 4.8 Updating ENS records . . . 17
- 4.9 Namehash . . . 18

5 ENS Library support **21**
- 5.1 Javascript . . . 21
- 5.2 Python . . . 21
- 5.3 Command-line . . . 21

6 Deploying ENS **23**
- 6.1 Deploy the registry . . . 23
- 6.2 Deploying a registrar . . . 24

7 Hosting a DNS domain on ENS **27**

8 FAQ **29**
- 8.1 Why are names registered as hashes? . . . 29
- 8.2 How do the DApp and the twitter bot know what names people are auctioning? . . . 29

ENS LABS EX1018
Page 22 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 22/59

8.3 What is the disguise amount / extra amount? . . . 29
8.4 Which wallets and DApps support ENS so far? . . . 30
8.5 Why does it say my name isn't available yet? . . . 30
8.6 How is the start time for each name determined? . . . 30
8.7 Why is my reveal transaction failing? . . . 30
8.8 I bid on an auction and didn't win; why don't I see the refund in my transaction list? . . . 30
8.9 I bid on one name, but Etherscan shows me as bidding on other name(s)! Why? . . . 31
8.10 Once I own a name, can I create my own subdomains? . . . 31
8.11 Can I change the address my name points to after I've bought it? . . . 31
8.12 How much do I have to deposit if I'm the only bidder? . . . 31
8.13 Can I register a TLD of my own in the ENS? . . . 31
8.14 Instead of burning funds in the auction for bidding costs and penalties, shouldn't they be donated to the Ethereum Foundation? . . . 31
8.15 Who will own the ENS rootnode? What powers does that grant them? . . . 32
8.16 What about foreign characters? What about upper case letters? Is any unicode character valid? . . . 32
8.17 Nameprep isn't enforced in the ENS system, is this a security/spoofing/phishing concern? . . . 32
8.18 How was the minimum character length of 7 chosen? . . . 32
8.19 What values will the permanent registrar try to optimize for? . . . 32
8.20 What kinds of behaviours are likely to result in losing ownership of a name? . . . 33
8.21 Can the bids be revealed during the auction period? . . . 33

9 Indices and tables 35

ENS LABS EX1018
Page 23 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 23/59

CHAPTER 1

Introduction



ENS is the Ethereum Name Service, a distributed, open, and extensible naming system based on the Ethereum blockchain.

ENS's job is to map human-readable names like 'alice.eth' to machine-readable identifiers such as Ethereum addresses, content hashes, and metadata. ENS also supports 'reverse resolution', making it possible to associate metadata such as canonical names or interface descriptions with Ethereum addresses.

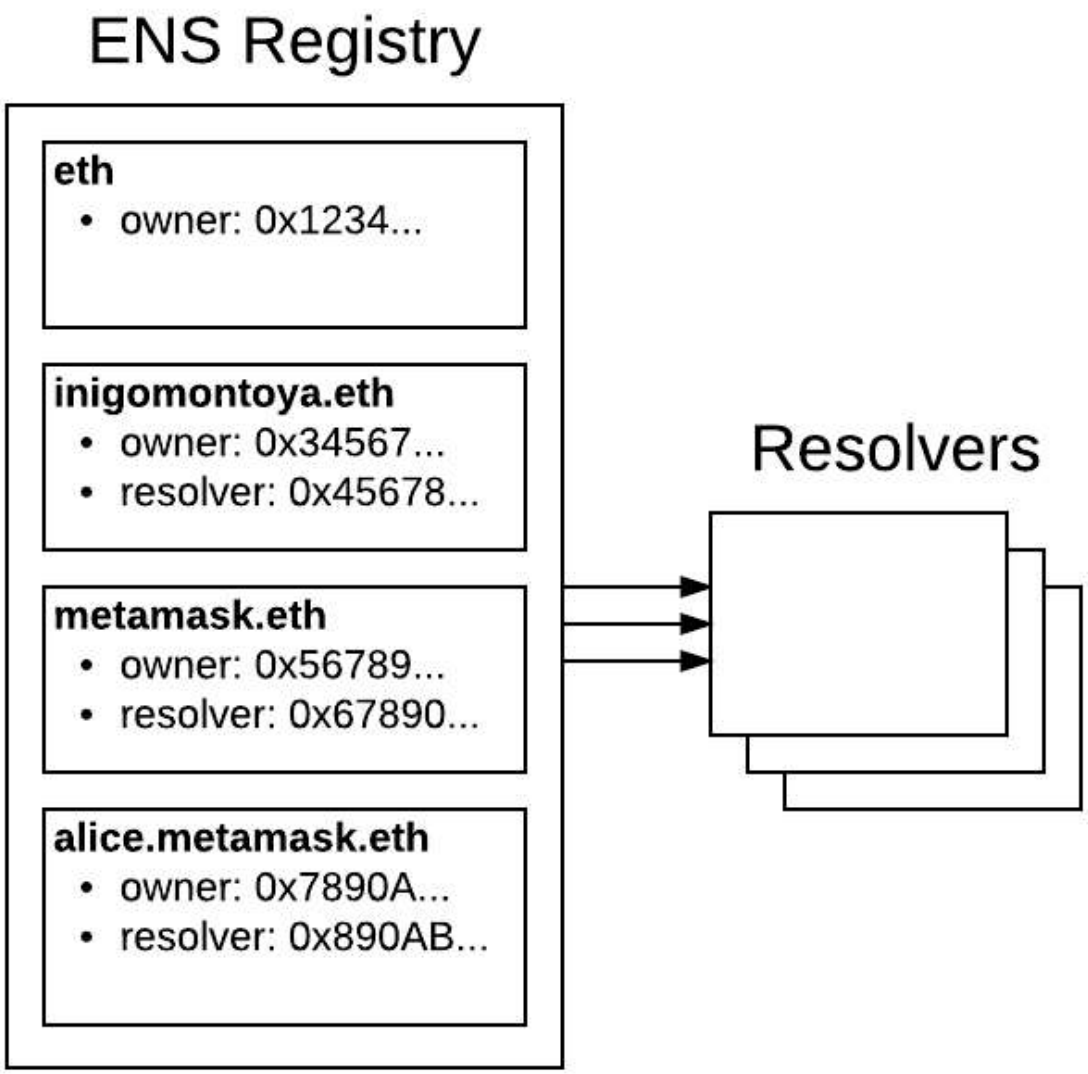
ENS has similar goals to DNS, the Internet's Domain Name Service, but has significantly different architecture, due to the capabilities and constraints provided by the Ethereum blockchain. Like DNS, ENS operates on a system of dot-separated hierarchial names called domains, with the owner of a domain having full control over the allocation of subdomains.

Top-level domains, like '.eth' and '.test' are owned by smart contracts called registrars, which specify rules governing the allocation of their subdomains. Anyone may, by following the rules imposed by these registrar contracts, obtain ownership of a second-level domain for their own use.

1.1 ENS in 5 minutes

ENS has two principal components: the registry, and resolvers.

ENS LABS EX1018
Page 24 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 24/59



The ENS registry consists of a single smart contract that maintains a list of all domains and subdomains, and stores three critical pieces of information about each:

- The owner of the domain
- The resolver for the domain
- The time-to-live for all records under the domain

The owner of a domain may be either an external account (a user) or a smart contract. A registrar is simply a smart contract that owns a domain, and issues subdomains of that domain to users that follow some set of rules defined in the contract.

Owners of domains in the ENS registry may:

- Set the resolver and TTL for the domain
- Transfer ownership of the domain to another address

ENS LABS EX1018
Page 25 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 25/59

- Change the ownership of subdomains

The ENS registry is deliberately straightforward, and exists only to map from a name to the resolver responsible for it.

Resolvers are responsible for the actual process of translating names into addresses. Any contract that implements the relevant standards may act as a resolver in ENS. General-purpose resolver implementations are offered for users whose requirements are straightforward, such as serving an infrequently changed address for a name.

Each record type - Ethereum address, Swarm content hash, and so forth - defines a method or methods that a resolver must implement in order to provide records of that kind. New record types may be defined at any time via the EIP standardisation process, with no need to make changes to the ENS registry or to existing resolvers in order to support them.

1.2 ENS on Ethereum

ENS is deployed on mainnet at 0x314159265dd8dbb310642f98f50c066173c1259b, where users may register names under the eth TLD, which uses an auction based registrar.

ENS is also deployed on the Ropsten testnet at 0x112234455c3a32fd11230c42e7bccd4a84e02010. Users may register names under two top level domains:

- .eth, which uses an auction based registrar with the same functionality as the main network, and allows users to keep names indefinitely; see *Registering a name with the auction registrar*.
- .test, which allows anyone to claim an unused name for test purposes, which expires after 28 days; see *Registering a name with the FIFS registrar*.

ENS is also deployed on the Rinkeby testnet at 0xe7410170f87102df0055eb195163a03b7f2bff4a, where only the .test top level domain is supported.

1.3 Resources

- EIP137 - Ethereum Name Service
- EIP162 - Initial ENS Registrar Specification
- ethereum-ens Javascript library
- Nick's talk on ENS at DevCon 2: https://www.youtube.com/watch?v=pLDDbCZXvTE
- DevCon 2 talk slides: https://arachnid.github.io/devcon2/#/title
- Web interface for Ethereum enabled browsers: https://registrar.ens.domains

ENS LABS EX1018
Page 26 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 26/59

ENS LABS EX1018
Page 27 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 27/59

CHAPTER 2

Quickstart

Just want to get a name and make it resolve to something? Here's how.

NOTE: This quickstart uses 'ensutils.js', a simple Javascript file that facilitates getting started with ENS. You should *NOT* use this in production code - it exists purely for interactive experimentation with the console!

First, make sure your client is in **sync with a network** (mainnet, ropsten, rinkeby, etc.). You can use eth.syncing in your Ethereum console to track progress.

You can utilize geth: https://github.com/ethereum/go-ethereum to connect a network and begin the process of syncing the chain.

Next, start an Ethereum console:

```
Mainnet: geth attach
Ropsten: geth --testnet attach
Rinkeby: geth --rinkeby attach
```

Download ensutils-testnet.js to your local machine, and import it into an Ethereum console on a node synced to ropsten or rinkeby:

```
loadScript('/path/to/ensutils-testnet.js'); // This will connect to the ropsten␣
↪testnet
```

If you want to use Rinkeby, you'll need to change in ensutils-testnet.js:

```
contract address: 0xe7410170f87102df0055eb195163a03b7f2bff4a (line 220)
publicResolver address: 0x5d20cf83cb385e06d2f2a892f9322cd4933eacdc (line 1314)
```

If you need to unlock your account to execute certain transactions:

```
// Careful as this will record your password in your history.
// You can unlock accounts with geth as well to avoid this.

web3.personal.unlockAccount(web3.personal.listAccounts[0],"<password>", 15000) // 15,
↪000 seconds
```


ENS LABS EX1018
Page 28 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 28/59

Before registering, check that nobody owns the name you want to register:

```
new Date(testRegistrar.expiryTimes(web3.sha3('myname')).toNumber() * 1000)
```

If this line returns a date earlier than the current date, the name is available and you're good to go. You can register the domain for yourself by running:

```
// This uses the .test registrar and will let you register the sub-domain myname␣
↪under .test
testRegistrar.register(web3.sha3('myname'), eth.accounts[0], {from: eth.accounts[0]})
```

Next, tell the ENS registry to use the public resolver for your name:

```
ens.setResolver(namehash('myname.test'), publicResolver.address, {from: eth.
↪accounts[0]});
```

Once that transaction is mined (you can use etherscan to do so), tell the resolver to resolve that name to your account:

```
publicResolver.setAddr(namehash('myname.test'), eth.accounts[0], {from: eth.
↪accounts[0]});
```

. . . or any other address:

```
publicResolver.setAddr(namehash('myname.test'), '0x1234...', {from: eth.accounts[0]});
```

If you want, create a subdomain and do the whole thing all over again:

```
ens.setSubnodeOwner(namehash('myname.test'), web3.sha3('foo'), eth.accounts[1],
↪{from: eth.accounts[0]});
ens.setResolver(namehash('foo.myname.test'), publicResolver.address, {from: eth.
↪accounts[1]});
...
```

Finally, you can resolve your newly created name:

```
getAddr('myname.test')
```

which is shorthand for:

```
resolverContract.at(ens.resolver(namehash('myname.test'))).addr(namehash('myname.test
↪'))
```


ENS LABS EX1018
Page 29 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 29/59

CHAPTER 3

User Guide

This user guide is intended for anyone wanting to register, configure, and update ENS names using a Javascript console and web3.js. Before starting, open up a geth console, download ensutils.js or ensutils-testnet.js to your local machine, and import it into a running Ethereum console:

```
loadScript('/path/to/ensutils.js');
```

3.1 Registering a name with the FIFS registrar

The public ENS deployment on Ropsten uses a first-in-first served registrar for the '.test' top level domain. Domains on this TLD are configured to expire, allowing anyone else to claim them, 28 days after registration.

ensutils.js defines an object *testRegistrar*, for interacting with the registrar for the *.test* TLD. If you want to interact with a different first-in-first-served registrar, you can instantiate it with:

```
var myRegistrar = fifsRegistrarContract.at(address);
```

Before registering, check that nobody owns the name you want to register:

```
new Date(testRegistrar.expiryTimes(web3.sha3('myname')).toNumber() * 1000)
```

If this line returns a date earlier than the current date, the name is available and you're good to go.

The FIFS registrar's interface is extremely simple, and exposes a method called *register* that you can call with the (hashed) name you want to register and the address you want to own the name. To register a name, simply call that method to send a registration transaction:

```
testRegistrar.register(web3.sha3('myname'), eth.accounts[0], {from: eth.accounts[0]});
```

Once this transaction is mined, assuming the name was not already assigned, it will be assigned to you, and you can proceed to *Interacting with the ENS registry*.

ENS LABS EX1018
Page 30 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 30/59

3.2 Registering a name with the auction registrar

Once deployed on mainnet, ENS names will be handed out via an auction process, on the '.eth' top-level domain. A preview of this is available on the Ropsten testnet, and you can register names via it right now. Any names you register will persist until launch on mainnet, at which point the auction registrar on Ropsten will be deprecated and eventually deleted.

This registrar implements a blind auction, and is described in EIP162. Names are initially required to be at least 7 characters long.

Registering a name with the auction registrar is a multi-step process.

3.2.1 Starting an auction

Before placing a bid, you need to check if the name is available. Run this code to check:

```
ethRegistrar.entries(web3.sha3('name'))[0];
```

This will return a single integer between 0 and 5. The full solidity data structure for this can be viewed here in the Registrar contract. The numbers represent different 'states' a name is currently in.

- 0 - Name is available and the auction hasn't started
- 1 - Name is available and the auction has been started
- 2 - Name is taken and currently owned by someone
- 3 - Name is forbidden
- 4 - Name is currently in the 'reveal' stage of the auction
- 5 - Name is not yet available due to the 'soft launch' of names.

If the returned value is *5*, and is in the 'soft launch' is in effect; you can check when it will be available for auction with:

```
new Date(ethRegistrar.getAllowedTime(web3.sha3('name')) * 1000);
```

To start an auction for a name that's not already up for auction, call *startAuction*:

```
ethRegistrar.startAuction(web3.sha3('name'), {from: eth.accounts[0], gas: 100000});
```

You can also start auctions for several names simultaneously, to disguise which name you're actually interested in registering:

```
ethRegistrar.startAuctions([web3.sha3('decoy1'), web3.sha3('name'), web3.sha3('decoy2
↪')], {from: eth.accounts[0], gas: 1000000});
```

Auctions normally run for 5 days: 3 days of bidding and 2 days of reveal phase. When initially deployed, there's a "soft start" phase during which names are released for bidding gradually; this soft start lasts 4 weeks on ropsten, and 13 weeks on mainnet.

When a name is under auction, you can check the end time of the auction as follows:

```
new Date(ethRegistrar.entries(web3.sha3('name'))[2].toNumber() * 1000)
```


ENS LABS EX1018
Page 31 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 31/59

3.2.2 Placing a bid

Bids can be placed at any time during an auction except in the last 48 hours (the 'reveal period'). Before trying to place a bid, make sure an auction is currently underway, as described above, and has more than 48 hours left to run.

To bid on an open auction, you need several pieces of data:

- The name you want to register
- The account you are sending the bid from
- The maximum amount you're willing to pay for the name
- A random 'salt' value

In addition, you need to decide how much Ether you want to deposit with the bid. This must be at least as much as the value of your bid, but can be more, in order to disguise the true value of the bid.

First, start by generating a secret value. An easy way to do this is to use random.org. Store this value somewhere secure - if you lose it, you lose your deposit, and your chance at winning the auction!

Now, you can generate your 'sealed' bid, with the following code:

```
var bid = ethRegistrar.shaBid(web3.sha3('name'), eth.accounts[0], web3.toWei(1, 'ether
↪'), web3.sha3('secret'));
```

The arguments are, in order, the name you want to register, the account you are sending the bid from, your maximum bid, and the secret value you generated earlier. Note that the bidding account will become the owner. You will lose funds if you seal with one account and send the bid with another!

Next, submit your bid to the registrar:

```
ethRegistrar.newBid(bid, {from: eth.accounts[0], value: web3.toWei(2, 'ether'), gas:␣
↪500000});
```

In the example above, we're sending 2 ether, even though our maximum bid is 1 ether; this is to disguise the true value of our bid. When we reveal our bid later, we will get the extra 1 ether back; the most we can pay for the name is 1 ether, as we specified when generating the bid.

Now it's a matter of waiting until the reveal period before revealing your bid. Run the command to check the expiration date of the auction again, and make sure to come back in the final 48 hours of the auction:

```
new Date(ethRegistrar.entries(web3.sha3('name'))[2].toNumber() * 1000)
```

3.2.3 Revealing your bid

In order to win an auction, you must 'reveal' your bid. This is only possible during the 'reveal' phase, the last 48 hours of the auction, at which point new bids are prohibited. If you don't reveal your bid by the time the auction ends, your deposit is forfeit - so make sure you store your salt in a safe place, and come back before the auction ends in order to reveal your bid.

To reveal, call the *unsealBid* function with the same values you provided earlier:

```
ethRegistrar.unsealBid(web3.sha3('name'), web3.toWei(1, 'ether'), web3.sha3('secret'),
↪ {from: eth.accounts[0], gas: 500000});
```

The arguments to *unsealBid* have the same order and meaning as those to *shaBid*, described in the bidding step, except that you don't need to supply the account - it's derived from your sending address.

After revealing your bid, the auction will be updated.

ENS LABS EX1018
Page 32 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 32/59

If your bid is less than a previously revealed bid, you will be refunded the whole amount of your bid.

If your bid is the largest revealed so far, you will be set as the current leading bidder. The difference between the actual amount of your bid and the amount you sent will be refunded immediately. The remainder - the actual bid - will stay locked. If you are later outbid it will be sent back to you.

3.2.4 Checking auctions

At any time, you can check the current winning bidder with:

```
deedContract.at(ethRegistrar.entries(web3.sha3('name'))[1]).owner();
```

and the value of the current winning bid with

```
web3.fromWei(ethRegistrar.entries(web3.sha3('name'))[4], 'ether');
```

3.2.5 Finalizing the auction

Once the auction has completed, it must be finalized in order for the name to be assigned to the winning bidder. Only the winning bidder can do this. To finalize, call the *finalizeAuction* function like so:

```
ethRegistrar.finalizeAuction(web3.sha3('name'), {from: eth.accounts[0], gas: 500000});
```

Once called, the winning bidder will be refunded the difference between their bid and the next highest bidder. If you're the only bidder, you get back all but 0.01 eth of your bid. The winner is then assigned the name in ENS.

If you are the winning bidder, congratulations!

3.2.6 Managing Ownership

After finalizing, your account now owns both the name in ENS and the deed in the Auction Registrar.

As the name owner, your account can manage the name using examples in "Interacting with the ENS registry". For example, you can use `ens.setOwner` to transfer administration of the name to another account. The new name owner can manage that domain and all subdomains now. None of those actions affect your ownership of the deed.

As the deed owner, your account has the right to reset name ownership back to itself at any time, by using `ethRegistrar.finalizeAuction` again. You can also choose to transfer the deed to another account with:

```
ethRegistrar.transfer(web3.sha3('name'), newDeedOwnerAddress, {from:␣
↪currentDeedOwnerAddress})
```

Caution: Transferring the deed is **irrevocable**. Be sure that you have verified the correct address for the new owner. Additionally, the ether you paid to win the auction will be transferred with the deed to the new owner.

3.3 Interacting with the ENS registry

The ENS registry forms the central component of ENS, mapping from hashed names to resolvers, as well as the owners of the names and their TTL (caching time-to-live).

ENS LABS EX1018
Page 33 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 33/59

Before you can make any changes to the ENS registry, you need to control an account that has ownership of a name in ENS. To obtain an ENS name on the Ropsten testnet, see *Registering a name with the auction registrar* for '.eth', or *Registering a name with the FIFS registrar* for '.test'. Names on '.test' are temporary, and can be claimed by someone else 28 days later.

Alternately, you can obtain a subdomain from someone else who owns a domain, or *Deploying ENS*. Note that while anyone can deploy their own ENS registry, those names will only be resolvable by users who reference that registry in their code.

3.3.1 Getting the owner of a name

You can retrieve the address of a name's owner using the *owner* function:

```
> ens.owner(namehash('somename.eth'));
"0xa303ddc620aa7d1390baccc8a495508b183fab59"
```

3.3.2 Getting the resolver for a name

You can retrieve the address of a name's resolver using the *resolver* function:

```
> ens.resolver(namehash('somename.eth'));
"0xc68de5b43c3d980b0c110a77a5f78d3c4c4d63b4"
```

3.3.3 Setting a name's resolver

You can set the resolver contract for a name using *setResolver*:

```
> ens.setResolver(namehash('somename.eth'), resolverAddress, {from: eth.accounts[0]});
```

A resolver is any contract that implements the resolver interface specified in EIP137. You can deploy your own resolver, or you can use a publicly available one; on the mainnet, a simple resolver that supports 'address' records and is usable by anyone is available; ensutils.js exposes it as *publicResolver*. To use it, first set it as the resolver for your name:

```
ens.setResolver(namehash('somename.eth'), publicResolver.address, {from: eth.
↪accounts[0]});
```

Then, call the resolver's *setAddr* method to set the address the name resolves to:

```
publicResolver.setAddr(namehash('somename.eth'), eth.accounts[0], {from: eth.
↪accounts[0]})
```

The above example configures 'somename.eth' to resolve to the address of your primary account.

3.3.4 Transferring a name

You can transfer ownership of a name you own in the ENS registry to another account using *setOwner*:

```
> ens.setOwner(namehash('somename.eth'), newOwner, {from: eth.accounts[0]});
```


ENS LABS EX1018
Page 34 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 34/59

Note: If the name was acquired through a registrar, such as through a '.eth' auction process, this will not transfer ownership of the locked bid. It will also not perform any administrative tasks that a registrar might want to do.

In general, to perform a complete transfer of a name acquired through a registrar, that particular registrar should be used as the interface. For the '.eth' registrar, see *Managing Ownership*.

3.3.5 Creating a subdomain

You can assign ownership of subdomains of any name you own with the *setSubnodeOwner* function. For instance, to create a subdomain 'foo.somename.eth' and set yourself as the owner:

```
> ens.setSubnodeOwner(namehash('somename.eth'), web3.sha3('foo'), eth.accounts[0],
↪{from: eth.accounts[0]});
```

Or, to assign someone else as the owner:

```
> ens.setSubnodeOwner(namehash('somename.eth'), web3.sha3('foo'), someAccount, {from:␣
↪eth.accounts[0]});
```

Note the use of *web3.sha3()* instead of *namehash()* when specifying the subdomain being allocated.

The owner of a name can reassign ownership of subdomains at any time, even if they're owned by someone else.

Note: When you first create your registry, the only domain is '.' (the root domain) and the root domain creates subdomains like the 'test' and 'eth' top-level domains.

3.3.6 Resolving Names

Now you're ready to resolve your newly created name. For details how, read *Resolving Names*.

3.3.7 Interacting with ENS from a DApp

An NPM module, ethereum-ens, is available to facilitate interacting with the ENS from Javascript-based DApps.

3.3.8 Interacting with ENS from a contract

The ENS registry interface provides a Solidity definition of the methods available for interacting with the ENS. Using this, and the address of the ENS registry, contracts can read and write the ENS registry directly.

A Solidity library to facilitate this will be available soon.

ENS LABS EX1018
Page 35 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 35/59

CHAPTER 4

Implementer's Guide

This section is intended to provide guidance for anyone wanting to implement tools and applications that use ENS, or custom resolvers within ENS.

4.1 3 steps to add ENS to your application

ENS integration in an application encompasses several critical features, each of which can be implemented independently. While comprehensive ENS integration is ideal, even basic support can be a huge benefit to users. Below, we outline the three main ways you can add ENS support to your application, starting with the simplest and highest impact.

4.1.1 1. Accept ENS names anywhere your app accepts addresses

The first step to supporting ENS in your application is making your application understand ENS names, and accepting them anywhere an address is accepted. To understand how to do this, see *Resolving Names*.

If possible, when a user enters an ENS name instead of an address, remember the ENS name, not the address it currently resolves to. This makes it possible for users to update their ENS names and have applications they used the name in automatically resolve to the new address, in the same way that you would expect your browser to automatically direct you to the new IP address if a site you use changes servers.

If your application deals with user funds or other critical resources, you may want to keep track of the address a name resolves to and warn them when it changes, to ensure they are aware of the change.

4.1.2 2. Display ENS names anywhere your app shows addresses

The second step is to display ENS names wherever your app currently displays addresses.

If a user entered an ENS name in step 1, you should retain this name and show it to them whenever you would normally show the address.

ENS LABS EX1018
Page 36 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 36/59

If a user entered an address, or the address was obtained from elsewhere, you may still be able to show an ENS name, by doing *Reverse Resolution*. This permits you to find the canonical name for an address and display that when possible. If no canonical name is provided, your application can fall back to displaying the address as it did previously.

4.1.3 3. Permit users to register names for their resources

The final step for comprehensive ENS integration is to facilitate associating ENS names with resources created by or managed with your application. This can take two forms:

1. Providing users with an easy way to register an ENS name and associate it with their resources (eg, wallet address); see *Writing a registrar*.
2. Providing users with an easy way to update a name they own to point at your application's resources; see *Updating ENS records*.

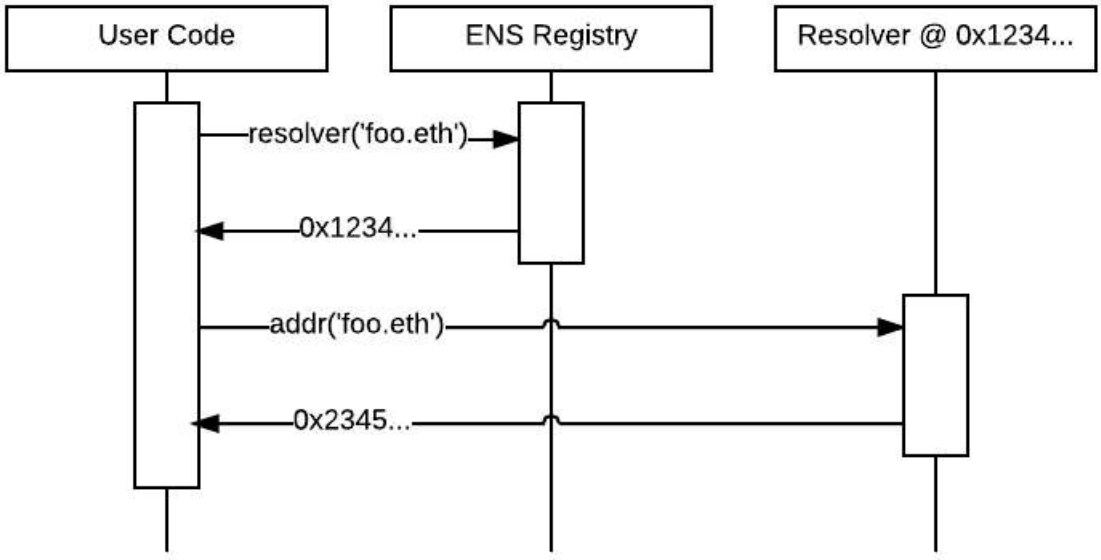
4.2 ENS libraries

The following sections describe how to interact with ENS directly. Chances are, there is already an ENS library for your language of choice, that will do the heavy lifting for you. To find one, see *ENS Library support*.

4.3 Resolving Names

Resolving names in ENS is a three step process:

1. Normalise and hash the name you want to resolve (see *Namehash*).
2. Query the ENS registry for the address of the resolver responsible for the name.
3. Query the resolver for the resource you want to look up.



ENS LABS EX1018
Page 37 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 37/59

To query the registry, call the *resolver(bytes32)* method on the ENS registry contract (see *ENS on Ethereum* for details on where to find the registry), passing it the namehash of the normalised name you are querying for. The registry will return the address of the resolver responsible for that name, or 0 if no resolver is configured or the name does not exist.

If the registry returned a nonzero resolver address, you can then query the resolver for the resource you want to resolve. For instance, to look up the Ethereum address associated with the name, you would call the resolver's *addr(bytes32)* method, again passing in the namehash. The resolver will then return the resource requested, if it's available.

A list of currently recognised resource types are defined in EIP137, but you can also define your own if you need. For details on how to do that, see *Writing a resolver*.

4.4 Reverse Resolution

In addition to mapping names to resources (forward resolution), ENS also supports mapping from addresses to names and other metadata (reverse resolution).

Reverse records are named *<ethereum address>.addr.reverse* - for instance, the official registry would have its reverse records at *314159265dd8dbb310642f98f50c066173c1259b.addr.reverse*.

Reverse resolution follows the same three-step process as forward resolution; the only change is that the name you are resolving has the special format defined above. To find the canonical name for an address, then, you would first query the ENS registry for the resolver responsible for *(address).addr.reverse*, then call *name(bytes32)* on that resolver in order to obtain its canonical name.

addr.reverse has a registrar with *claim(address)*, *claimWithResolver(address, address)*, and *setName(string)* functions.

The claim function takes one argument, the Ethereum address that should own the reverse record.

This permits a very simple pattern for contracts that wish to delegate control of their reverse record to their creator; they simply need to add this function call to their constructor:

```
reverseRegistrar.claim(msg.sender)
```

External accounts can also call this function to claim ownership of their reverse record. For simplicity, the *setName(string)* function permits claiming a record and configuring a canonical name all in one step.

4.5 Writing a resolver

Resolvers are specified in EIP137. A resolver must implement the following method:

```
function supportsInterface(bytes4 interfaceID) constant returns (bool)
```

supportsInterface is defined in EIP165, and allows callers to determine if a resolver supports a particular record type. Record types are specified as a set of one or more methods that a resolver must implement together. Currently defined record types include:

Record type	Function(s)	Interface ID	Defined in
Ethereum address	*addr*	0x3b3b57de	EIP137
ENS Name	*name*	0x691f3431	EIP181
ABI specification	*ABI*	0x2203ab56	EIP205
Public key	*pubkey*	0xc8690233	EIP619

supportsInterface must also return true for the *interfaceID* value *0x01ffc9a7*, which is the interface ID of *supportsInterface* itself.

ENS LABS EX1018
Page 38 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 38/59

Additionally, the *content()* interface is currently used as a defacto standard for Swarm hashes, pending standardisation, and has an interface ID of *0xd8389dc5*.

For example, a simple resolver that supports only the *addr* type might look something like this:

```
contract SimpleResolver {
    function supportsInterface(bytes4 interfaceID) constant returns (bool) {
        return interfaceID == 0x3b3b57de;
    }

    function addr(bytes32 nodeID) constant returns (address) {
        return address(this);
    }
}
```

This trivial resolver always returns its own address as answer to all queries. Practical resolvers may use any mechanism they wish to determine what results to return, though they should be *constant*, and should minimise gas usage wherever possible.

NOTE: If you are resolving *addr()* records, you MUST treat a return value from the resolver of *0x00...00* as that record being unset. Failing to do so could result in users accidentally sending funds to the null address if they have configured a resolver in ENS, but not set the resolver record!

4.6 Resolving names onchain

Solidity libraries for onchain resolution are not yet available, but ENS resolution is straightforward enough it can be done trivially without a library. Contracts may use the following interfaces:

```
contract ENS {
    function owner(bytes32 node) constant returns (address);
    function resolver(bytes32 node) constant returns (Resolver);
    function ttl(bytes32 node) constant returns (uint64);
    function setOwner(bytes32 node, address owner);
    function setSubnodeOwner(bytes32 node, bytes32 label, address owner);
    function setResolver(bytes32 node, address resolver);
    function setTTL(bytes32 node, uint64 ttl);
}

contract Resolver {
    function addr(bytes32 node) constant returns (address);
}
```

For resolution, only the *resolver()* function in the ENS contract is required; other methods permit looking up owners, and updating ENS from within a contract that owns a name.

With these definitions, looking up a name given its node hash is straightforward:

```
contract MyContract {
    ENS ens;

    function MyContract(address ensAddress) {
        ens = ENS(ensAddress);
    }

    function resolve(bytes32 node) constant returns(address) {
        var resolver = ens.resolver(node)
```

(continues on next page)

ENS LABS EX1018
Page 39 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 39/59

(continued from previous page)

```
        return resolver.addr(node);
    }
}
```

While it is possible for a contract to process a human-readable name into a node hash, we highly recommend working with node hashes instead, as they are easier to work with, and allow contracts to leave the complex work of normalising the name to their callers outside the blockchain. Where a contract always resolves the same names, those names may be converted to a node hash and stored in the contract as a constant.

4.7 Writing a registrar

A registrar in ENS is simply any contract that owns a name, and allocates subdomains of it according to some set of rules defined in the contract code. A trivial first in first served contract is demonstrated below, using the ENS interface definition defined earlier.

```
contract FIFSRegistrar {
    ENS ens;
    bytes32 rootNode;

    function FIFSRegistrar(address ensAddr, bytes32 node) {
        ens = ENS(ensAddr);
        rootNode = node;
    }

    function register(bytes32 subnode, address owner) {
        var node = sha3(rootNode, subnode);
        var currentOwner = ens.owner(node);

        if (currentOwner != 0 && currentOwner != msg.sender) throw;

        ens.setSubnodeOwner(rootNode, subnode, owner);
    }
}
```

You may wish to set custom rules for the allocation of new names to your users; the rules you set are entirely up to you.

You should also bear in mind that as long as you retain ownership of the parent name - either directly or through another contract - your users have no guarantee that you will not take back ownership of their names and change what they resolve to. You may wish to consider committing ownership of the name to a contract that restricts your ability to control it. For an example of this, see ENSNow.

4.8 Updating ENS records

Your application may wish to provide users with a means of updating names they own to point to resources your application provides or manages. Doing so follows a similar process to *Resolving Names*:

1. Normalise and hash the name you want to resolve (see *Namehash*).
2. Query the ENS registry for the address of the resolver responsible for the name.
3. Call the appropriate update method on the resolver.

ENS LABS EX1018
Page 40 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net



Steps 1 and 2 are the same as in *Resolving Names*. If step 2 fails to return a valid resolver address, you will need to inform your user and offer them alternatives - for instance, you can check if they own the name in question, and if they do, offer to configure a default resolver for them by calling *setResolver* on the ENS registry.

In step 3, call the method defined by the resolver profile - standard profiles are listed in EIP137. For instance, to set the address associated with the name, call *setAddr(bytes32, address)*, passing in the name hash of the name to update and the address you wish it to point to.

4.9 Namehash

Names in ENS are represented as 32 byte hashes, rather than as plain text. This simplifies processing and storage, while permitting arbitrary length domain names, and preserves the privacy of names onchain. The algorithm used to translate domain names into hashes is called namehash. The Namehash algorithm is defined in EIP137.

In order to preserve the hierarchal nature of names, namehash is defined recursively, making it possible to derive the hash of a subdomain from the namehash of the parent domain and the name or hash of the subdomain label.

4.9.1 Terminology

- **domain** - the complete, human-readable form of a name; eg, *'vitalik.wallet.eth'*.
- **label** - a single component of a domain; eg, *'vitalik'*, *'wallet'*, or *'eth'*. A label may not contain a period ('.').
- **label hash** - the output of the keccak-256 function applied to a label; eg, *keccak256('eth')* = *0x4f5b812789fc606be1b3b16908db13fc7a9adf7ca72641f84d75b47069d3d7f0*.
- **node** - the output of the namehash function, used to uniquely identify a name in ENS.

4.9.2 Algorithm

First, a domain is divided into labels by splitting on periods ('.'). So, 'vitalik.wallet.eth' becomes the list ['vitalik', 'wallet', 'eth'].

The namehash function is then defined recursively as follows:

```
namehash([]) = 0x0000000000000000000000000000000000000000000000000000000000000000
namehash([label, ...]) = keccak256(namehash(...), keccak256(label))
```

A sample implementation in Python is provided below.

```
def namehash(name):
  if name == '':
    return '\0' * 32
  else:
    label, _, remainder = name.partition('.')
    return sha3(namehash(remainder) + sha3(label))
```

4.9.3 Normalising and validating names

Before a name can be converted to a node hash using Namehash, the name must first be normalised and checked for validity - for instance, converting *fOO.eth* into *foo.eth*, and prohibiting names containing forbidden characters such as underscores. It is crucial that all applications follow the same set of rules for normalisation and validation, as otherwise

ENS LABS EX1018
Page 41 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 41/59

two users entering the same name on different systems may resolve the same human-readable name into two different ENS names.

Applications using ENS and processing human-readable names must follow UTS46 for normalisation and validation. Processing should be done with non-transitional rules, and with *UseSTD3ASCIIRules=true*.

The ethereum-ens Javascript library incorporates compliant preprocessing into its *validate* and *namehash* functions, so users of this library avoid the need to handle this manually.

4.9.4 Handling of ambiguous names

Because of the large number of characters in unicode, and the wide variety of scripts represented, inevitably there are different Unicode characters that are similar or even identical when shown in common fonts. This can be abused to trick users into thinking they are visiting one site or resource, when in fact they are visiting another. This is known as a homoglyph attack.

User agents and other software that display names to users should take countermeasures against these attacks, such as by highlighting problematic characters, or showing warnings to users about mixed scripts. Chromium's IDNA strategy may serve as a useful reference for user-agent behaviour around rendering IDNA names.

ENS LABS EX1018
Page 42 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 42/59

ENS LABS EX1018
Page 43 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 43/59

CHAPTER 5

ENS Library support

This page lists libraries that support ENS in different languages, for your convenience. If you know of a library that is not listed here, please send us a PR.

5.1 Javascript

- ethereum-ens
- ethjs-ens

5.2 Python

- ens.py
- web3.py

5.3 Command-line

- ethers-ens

ENS LABS EX1018
Page 44 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 44/59

ENS LABS EX1018
Page 45 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 45/59

CHAPTER 6

Deploying ENS

If you'd like to deploy ENS on your own network, or deploy your own copy of ENS on a public network, this guide shows you how. If you want to use an existing ENS deployment, read *Interacting with the ENS registry* instead. If you want to register a name on the Ropsten (testnet) ENS deployment, read *Registering a name with the FIFS registrar* or *Registering a name with the auction registrar*.

6.1 Deploy the registry

First, you need to deploy ENS's central component, the registry. To do so, paste this code into an Ethereum console:

ENS LABS EX1018
Page 46 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 46/59

```
var ensContract = web3.eth.contract([{"constant":true,"inputs":[{"name":"node","type":
↪"bytes32"}],"name":"resolver","outputs":[{"name":"","type":"address"}],"payable
↪":false,"type":"function"},{"constant":true,"inputs":[{"name":"node","type":"bytes32
↪"}],"name":"owner","outputs":[{"name":"","type":"address"}],"payable":false,"type":
↪"function"},{"constant":false,"inputs":[{"name":"node","type":"bytes32"},{"name":
↪"label","type":"bytes32"},{"name":"owner","type":"address"}],"name":"setSubnodeOwner
↪","outputs":[],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name
↪":"node","type":"bytes32"},{"name":"ttl","type":"uint64"}],"name":"setTTL","outputs
↪":[],"payable":false,"type":"function"},{"constant":true,"inputs":[{"name":"node",
↪"type":"bytes32"}],"name":"ttl","outputs":[{"name":"","type":"uint64"}],"payable
↪":false,"type":"function"},{"constant":false,"inputs":[{"name":"node","type":
↪"bytes32"},{"name":"resolver","type":"address"}],"name":"setResolver","outputs":[],
↪"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"node","type
↪":"bytes32"},{"name":"owner","type":"address"}],"name":"setOwner","outputs":[],
↪"payable":false,"type":"function"},{"anonymous":false,"inputs":[{"indexed":true,
↪"name":"node","type":"bytes32"},{"indexed":false,"name":"owner","type":"address"}],
↪"name":"Transfer","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name
↪":"node","type":"bytes32"},{"indexed":true,"name":"label","type":"bytes32"},{
↪"indexed":false,"name":"owner","type":"address"}],"name":"NewOwner","type":"event"},
↪{"anonymous":false,"inputs":[{"indexed":true,"name":"node","type":"bytes32"},{
↪"indexed":false,"name":"resolver","type":"address"}],"name":"NewResolver","type":
↪"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"node","type":"bytes32
↪"},{"indexed":false,"name":"ttl","type":"uint64"}],"name":"NewTTL","type":"event"}
↪]);
var ens = ensContract.new({
    from: web3.eth.accounts[0],
    data:
↪"0x336000600015561022080610014600039600f3630178b8bf60e060020a60003504141561002357602060043501546(
↪",
    gas: 4700000
}, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + '␣
↪transactionHash: ' + contract.transactionHash);
    }
});
```

Once successfully mined, you will have a fresh ENS registry, whose root node is owned by the account that created the transaction (in this case, the first account on your node). This account has total control over the ENS registry - it can create and replace any node in the entire tree.

For instructions on how to interact with the registry, see *Interacting with the ENS registry*.

6.2 Deploying a registrar

A registrar is a contract that has ownership over a node (name) in the ENS registry, and provides an interface for users to register subnodes (subdomains). You can deploy a registrar on any name; in this example we'll deploy a simple first-in-first-served registrar for the root node.

To deploy a first-in-first-served registrar on the root node of an ENS registry you control, execute this code in an Ethereum console:

```
var registrarContract = web3.eth.contract([{"constant":false,"inputs":[{"name":
↪"subnode","type":"bytes32"},{"name":"owner","type":"address"}],"name":"register",
↪"outputs":[],"payable":false,"type":"function"},{"inputs":[{"name":"ensAddr","type":
↪"address"},{"name":"node","type":"bytes32"}, {"name": "_startDate", "type": "uint256
↪"}],"type":"constructor"}]);
```

(continues on next page)

ENS LABS EX1018
Page 47 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 47/59

(continued from previous page)

```
var registrar = registrarContract.new(
    ens.address,
    0,
    0,
    {from: web3.eth.accounts[0],
    data:
↪"0x6060604081815280610c4833960a090525160805160008054600160a060
↪",
    gas: 4700000
}, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + '␣
↪transactionHash: ' + contract.transactionHash);
    }
});
```

Once that transaction is mined, you can transfer ownership of the root node to the newly created registrar:

```
ens.setOwner(0, registrar.address, {from: web3.eth.accounts[0]});
```

Users can now register names with the registrar; for instructions read *Registering a name with the FIFS registrar*.

ENS LABS EX1018
Page 48 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net

ENS LABS EX1018
Page 49 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 49/59

CHAPTER 7

Hosting a DNS domain on ENS

Experimental support is available for hosting DNS domains on the Ethereum blockchain via ENS. This works by configuring the domain's nameserver records to point to gateway DNS servers; these DNS servers resolve lookups by consulting an ENS registry which points to resolvers containing the zone data for the relevant domain.

The steps to host your own domain on the blockchain are:

1. deploy your own ENS registry
2. Deploy an instance of DNSResolver
3. Update your ENS registry to set your newly deployed DNSResolver as the resolver for your domain name (eg, by calling setSubnodeOwner and setResolver; see *Interacting with the ENS registry* for details). Don't forget to set the TTL on this record to a reasonable value, or your DNS data may not get cached.
4. Write a zonefile. The zonefile must include an NS record for your domain that specifies the resolver as *address*.ns1.ens.domains, where *address* is the address of the ENS registry you deployed in step 1, without the leading '0x'. An example zonefile is available for reference.
5. Clone and build ensdns. Start a local node, and run 'ensdns upload –keystore=path/to/keystore zonefile' to upload the zone to the blockchain.
6. Update your NS records with your registrar to point to the name above (*address*.ns1.ens.domains).

Please note that this feature is still experimental, and shouldn't be used for anything production critical; the DNS gateway is lightly tested, and only a single instance is running at present, providing no backup or failover in case of server issues. The API and configuration may change in backwards-incompatible ways, breaking your nameserver!

ENS LABS EX1018
Page 50 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 50/59

ENS LABS EX1018
Page 51 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 51/59

CHAPTER 8

FAQ

8.1 Why are names registered as hashes?

Hashes are used for two main reasons. First, to prevent trivial enumeration of the entire set of domains, which helps preserve privacy of names (for instance, so you can register the domain for your startup before you publicly launch). Second, because hashes provide a fixed length identifier that can easily be passed around between contracts with fixed overhead and no issues around passing around variable-length strings.

8.2 How do the DApp and the twitter bot know what names people are auctioning?

The DApp and the twitter bot have built in lists of common names, drawn from an English dictionary and Alexa's list of top 1 million internet domain names. They use these lists to show you when common names are being auctioned. We do this because if the app didn't reveal these names, anyone with a little technical skill could find them out anyway, giving them an advantage over those who don't have the capacity to build their own list and code to check names against it.

8.3 What is the disguise amount / extra amount?

Although it's difficult for someone to determine what name you are bidding on, in some circumstances it's possible to either determine the exact name, or narrow it down to one of several. The disguise amount allows you to send extra ether along with your bid, to disguise the true amount of your bid. This ether is returned to you as soon as you reveal your bid.

ENS LABS EX1018
Page 52 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 52/59

8.4 Which wallets and DApps support ENS so far?

MyEtherWallet supports both registering names via the auction process and sending funds and interacting with contracts identified by their names.

Metamask supports sending funds to ENS names.

Mist is working on ENS support and should announce it soon.

LEth is working on ENS support and should announce it soon.

Status is working on ENS support and should announce it soon.

8.5 Why does it say my name isn't available yet?

ENS names are released gradually over a 'slow start' period of 8 weeks starting on May 4th 2017. The time at which any given name becomes available for auction during that period is effectively random. If you enter your desired name into the DApp, it will let you know when the first time you can auction it is.

8.6 How is the start time for each name determined?

Internally, we hash the name using keccak256, and express the result as a number between 0 and 1. Then, we multiply that by the duration of the launch period (8 weeks) and add that to the start date (May 4th 2017 1100 UTC) to generate the time at which that name can first be auctioned. You can see the code for this here.

8.7 Why is my reveal transaction failing?

Confirm all the values you put in during the bid, and try using a blockchain explorer to confirm the state of the ENS auction. These are situations that cause errors during the reveal transaction:

- You never successfully bid
- You're trying to reveal too early
- The domain name is wrong
- The amount is wrong
- The secret phrase is wrong
- You already revealed

8.8 I bid on an auction and didn't win; why don't I see the refund in my transaction list?

Refunds are sent as part of reveal transactions - when you reveal, if someone already outbid you, or when you're outbid later, if not - and as part of finalise transactions. They're not separate transactions, so they don't show up as such. If you're using etherscan, check the 'internal transctions' tab to find your refund information.

ENS LABS EX1018
Page 53 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 53/59

8.9 I bid on one name, but Etherscan shows me as bidding on other name(s)! Why?

The registrar DApp opens up to three randomly selected auctions as 'chaff' every time you place a bid, to make it more difficult for people to guess what you are bidding on. The names you see are not necessarily the one you're bidding on; that's contained in the sealed bid hash that was submitted.

8.10 Once I own a name, can I create my own subdomains?

Yes! You can create whatever subdomains you wish, and assign ownership of them to other people if you desire. You can even set up your own registrar for your domain!

8.11 Can I change the address my name points to after I've bought it?

Yes, you can update the addresses and other resources pointed to by your name at any time.

8.12 How much do I have to deposit if I'm the only bidder?

If only one bid is revealed, that bidder deposits 0.01 ether, the minimum bid. The remainder of their bid is refunded as soon as they finalise the auction.

8.13 Can I register a TLD of my own in the ENS?

No, TLDs are restricted to only .eth (on mainnet), or .eth and .test (on Ropsten), plus any special purpose TLDs such as those required to permit reverse lookups. There are no immediate plans to invite proposals for additional TLDs. In large part this is to reduce the risk of a namespace collision with the IANA DNS namespace.

8.14 Instead of burning funds in the auction for bidding costs and penalties, shouldn't they be donated to the Ethereum Foundation?

Burning is fairly rare in the current registrar; it only burns fees if you reveal an illegal bid, or fail to reveal a bid during the reveal period. In all other circumstances they're refunded to users, either when you're outbid or when you relinquish the name. A small portion (0.5%) of the bids are burned with the intent of creating a cost for a large amount of domains or for highly valuable domains without the intention of buying them.

Burning fees is impartial, and avoids both political concerns over the destination of the funds, and perverse incentives for the beneficiary of the fees. The value of the ether burned is not destroyed, but rather equally distributed among all ether holders.

ENS LABS EX1018
Page 54 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 54/59

8.15 Who will own the ENS rootnode? What powers does that grant them?

The root node will initially be owned by a multisig contract, with keys held by trustworthy individuals in the Ethereum community. The exact makeup of this has not yet been decided on. We expect that this will be very hands-off, with the root ownership only used to effect administrative changes, such as the introduction of a new TLD, or to recover from an emergency such as a critical vulnerability in a TLD registrar.

In the long term, the plan is to define a governance process for operations on the root node, and transfer ownership to a contract that enforces this process.

Since the owner of a node can change ownership of any subnode, the owner of the root can change any node in the ENS tree.

8.16 What about foreign characters? What about upper case letters? Is any unicode character valid?

Since the ENS contracts only deal with hashes, they have no direct way to enforce limits on what can be registered; character length restrictions are implemented by allowing users to challenge a short name by providing its preimage to prove it's too short.

This means that you can in theory register both 'foo.eth' and 'FOO.eth', or even <picture of my cat>.eth. However, resolvers such as browsers and wallets should apply the nameprep algorithm to any names users enter before resolving; as a result, names that are not valid outputs of nameprep will not be resolvable by standard resolvers, making them effectively useless. DApps that assist users with registering names should prevent users from registering unresolvable names by using nameprep to preprocess names being requested for registration.

8.17 Nameprep isn't enforced in the ENS system, is this a security/spoofing/phishing concern?

It's not enforced by the ENS contracts, but as described, resolvers are expected to use it before resolving names. This means that non-nameprep names will not be resolvable.

8.18 How was the minimum character length of 7 chosen?

By an informal survey of common 'high value' short names. This restriction is intended to be lifted once the permanent registrar is in place.

8.19 What values will the permanent registrar try to optimize for?

This is something that the community will have to decide as part of the standardisation process for the permanent registrar. A few possible principles to consider include:

- Accessibility: Registering a new name should be as easy and straightforward as possible.
- Correct valuation: registering a known or popular name should be costly and intentional, not a matter of luck
- Fairness: The system should not unduly favor people who happen to be in the right place at the right time.

ENS LABS EX1018
Page 55 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 55/59

- Stability: Names should only be reallocated with the express will of the owner or according to objective rules that will be discussed and set with the whole community.
- Principle of least surprise: Wherever possible, names should resolve to the resource most users would expect it to resolve to.

8.20 What kinds of behaviours are likely to result in losing ownership of a name?

This is the most important aspect to be decided on the Permanent registrar and the one we want more open debate. At minimum we want the owner of a name to have to execute some periodical transaction, just to prove that name hasn't been abandoned or keys have been lost. This transaction would probably also require additional ether to be locked or burned. The method to which that amount is calculated is yet to be determined but would probably be dependent on some (but not necessarily all) of these factors:

- The amount of ethers the domain was bought for originally
- The average cost of a domain back when it was first bought
- The average cost of a domain at the moment of renewal
- The current market value of the domain (to be calculated with some auction method)
- Other factors to be discussed

Just like the current model, this "fee" would not go to the Ethereum Foundation or any third party, but be locked or burned. Ideally, this financial (opportunity and liquidity) cost will make name squatting unprofitable – or at least make the name reselling market a dynamic and competitive one, focused on quick turnout and not on holding names long term for as much money as possible.

Another very possible option creating some sort of dispute resolution process for names, to ensure the "principle of least surprise" but this is a controversial idea and there are no clear ideas on how this process could be achieved in a fair way without risks of centralization and abuse of power.

8.21 Can the bids be revealed during the auction period?

No, you can't reveal during the auction period, only during the reveal period. Bids revealed during the auction period will have no effect and wouldn't count for the reveal period.

This document is licensed under the @emph{Creative Commons Attribution License}. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/

ENS LABS EX1018
Page 56 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net



ENS LABS EX1018
Page 57 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 57/59

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

ENS LABS EX1018
Page 58 of 59
Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net
Page 58/59

JURAT ATTACHMENT

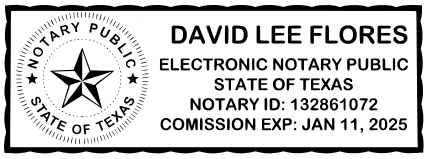
A notary public or other officer completing this certificate verifies only the identity of the individual who signed the document to which this certificate is attached, and not the truthfulness, accuracy, or validity of that document.

STATE OF Texas }

COUNTY OF Hidalgo }

The foregoing instrument was subscribed and sworn before me this date of 01/27/2024, by Nathaniel Frank-White

This notarial act was an online notarization.



(Notary Seal)

Notary's Signature _____

Registration No.: 132861072

Commission Expiration Date: January 11, 2025

Document Notarized using a Live Audio-Video Connection

Document Id: D6470660-BD7C-11EE-B62E-43A003CE0C51
OnlineNotary.net