An Authentication Service for Domestic Self-Hosting

F.T. Breggeman BSc

Abstract—Domestic self-hosting refers to the practice of operating and maintaining a server for personal use. It has been found that there are no authentication services that are suitable for use in domestic self-hosting. This paper aims to rectify this by investigating what the challenges are in creating such a service, with DAS, an authentication service that is suitable for domestic self-hosting, being an artifact produced by this research.

Index Terms—self-hosting, domestic self-hosting, authentication, OAuth, LDAP, SAML, Forward Authentication, Reverse Proxy Authentication, PAM

1 Introduction

In October of 2021, a survey was held on the most prominent communities for domestic self-hosting[7]. This survey is, to the best of my knowledge, the first academic source on domestic self-hosting. In order to gain more knowledge on domestic-self hosting, and specifically software development for domestic self-hosting, the idea was conceived of developing a program specifically for use in domestic self-hosting, and documenting the process and the results. The survey revealed that only relatively few domestically self-hosted systems have any form of centralised authentication service, and those that are rarely integrated with all services on the system. As such, the project became focussed on creating an authentication provider for specific use by domestic self-hosting; this project would be named the Domestic Authentication Service, or DAS.

The survey identified the most popular software that is domestically self-hosted, and the authentication protocols that they support (if any). The paper identified that there is no one standard, but that most software that supports any sort of centralised authentication uses either OAuth, LDAP, or reverse-proxy authentication, with SAML and PAM being less common. As such, any software aiming to support domestic-self hosting should be compatible with at least the three most common protocols, with SAML and PAM being optional features. This paper describes the challenges in designing and implementing such a service, both in the supporting multiple protocols, but especially in designing software for self-hosting. In order to do so, it defines the following research questions:

1) What are the differences between OAuth, LDAP, Reverse Proxy, SAML, and PAM?
2) What are the challenges involved in making a service that can natively support multiple protocols, most notably OAuth, LDAP, and reverse proxy?
3) What are the challenges involved in ensuring such a service is suitable for domestic self-hosting?
4) What are the differences between a self-hosted authentication service and its enterprise counterparts?
5) What authentication schemes (e.g. traditional username/password, TOTP, asymmetric) can operate within the commonly used protocols?

This paper is divided into 8 sections, including this one.

Section 2 gives some context on the two main areas of this paper, domestic self-hosting and authentication. Section 3 identifies and briefly describes some related work. Section 4 describes each of the protocols in details, provides relevant information for the implementation of the service, and makes a comparison of the various protocols. In doing so, it answers research question 1. Section 4.8 answers research question 5. Section 5 is a software architecture of the service. It contributes to answering questions 2, 3, and 4. Section 6 describes any relevant implementation details that do not follow from the architecture. It primaliry answers research questions 2, but also contributes to answering research questions 3 and 4. Section 7 contains summarised answers to each of the research questions, and refers to the place in this paper where more detailed answers can be found. Section 8 concludes this paper.

1.1 Terminology

To avoid confusion, the following definitions will be used throughout this document unless explicitly specified otherwise.

- **Authentication**: The process of proving the identity of a user to an application.
- **End user**: A natural person who wants to use a service (and needs to authenticate themselves in order to do so)
- **Credentials**: Some data that can be used to uniquely identify and authenticate an end user.
- **Authentication provider**: A service which authenticates end users for applications other than itself.
- **Client application**: An application which has users that are authenticated by the corresponding authentication provider

- **Authentication scheme**: A method by which an authentication provider or other application can verify the identify of a user
- **Authentication protocol**: A method by which an authentication provider can securely inform a client application of the identity of a user

2 Background

2.1 Domestic self-hosting

This section offers a brief description of domestic self-hosting. Statements in this section are roughly based on the results from the aforementioned survey[7].

Self-hosting can be defined as the practice of operating a server which provides software services for use by (or otherwise directly beneficial to) the operator. For example, a small company might host their own website, or a museum might host their own organisational software[62]. This paper focuses on domestic self-hosting: hosting software for personal use, traditionally from a server in one's own home. Examples of services that are commonly self-hosted domestically include email, personal websites, cloud storage, instant messaging, and media streaming.

Domestic self-hosting should be considered an alternative to cloud-based services, e.g. Microsoft Outlook, Teams, and OneDrive. These cloud-based services offer similar functionality to many self-hosted programs, e.g. Dovecot[16]/Postfix[46], Matrix[36], and NextCloud[39]. Self-hosting these services means one has more control over their data, leading to greater privacy and less dependence of external parties. Self-hosting may allow one to use software, or certain features of particular software, that is not available or prohibitively expensive in cloud-based services. Finally, self-hosting could be cheaper as the software is mostly open-source or shareware; this holds especially true if one wants large amounts of storage or other hardware resources. Of course, domestic self-hosting also has disadvantages. Because self-hosting gives one responsibility over the services, it also means that one will have to put effort into maintaining the services, and if they break one will have to perform repair. Unlike cloud-based services, there is very little customer support available for self-hosted services, although the communities are usually willing to help. Finally, the lower adoption rate for and higher barriers to entry to self-hosted services compared to their cloud-based counterparts can form a disadvantage for communication or collaboration services, e.g. Matrix[36] or Collabora[32], where it is expected that a group of people working together all have an account at the same service.

Self-hosting requires some degree of technical knowledge. At the very basics, a rudimentary understanding of server-client architectures is required. Basic networking knowledge is also required, including some knowledge on IP addresses, DNS, and firewalls. Working from behind a residential connection requires port forwarding, and sometimes also assigning a static IP address. Some server management skills are often used, as many users configure the software they host manually; however, there does exist software which aims to automate this[63].

Unique to domestic self-hosting is that servers are often physically located in a private residence, and have to deal with the limitations of consumer internet, such as low upload speeds and ISP-provided NATs, as opposed to the dedicated server infrastructure used in enterprise hosting. The hardware used for domestic self-hosting can range from small, low-power devices such as the Raspberry Pi to enterprise-grade server racks. Older hardware, ranging from discarded desktops to deprecated enterprise hardware, is quite commonly used. Especially the inclusion of older hardware is quite distinct from enterprise hosting, where hardware will almost always be purchased new and taken out of service after a few years, to take advantage of the better performance and efficiency of newer hardware. The software can also be quite distinct; software that is self-hosted by enterprises is often expected to be used by multiple distinct groups of users, and will generally have complex access control mechanisms to mirror the complex structure of a company. In domestic self-hosting, however, there is expectation for only few users, with at most a distinction between users who are administrators and users who are not. Interoperability is also different; most enterprise software will either be part of a vendor ecosystem, or integrate with one or multiple such ecosystems. In domestic self-hosting, these ecosystems do not exist. Instead, software will usually attempt to support a standard wherever integration is desired. While this usually leads to compatibility, it can lead to problems when there are multiple standards without an accepted default.

2.2 Authentication

One of the areas in which there multiple common standards is authentication; specifically, there are multiple protocols that allow a service to check credentials at a centralised location. In domestic self-hosting, LDAP, OAuth and reverse proxy authentication are most common, with some services also supporting SAML or PAM. However, none of these protocols is *the* standard, and any reasonably sized self-hosted system will find that no protocol is supported by all of the hosted services. As such, any centralised authentication provider for domestic self-hosting must support multiple protocols. There are currently two open source authentication providers that have support for multiple protocols.

Keycloak[27] is an open source authentication provider written in Java. It natively supports OAuth and SAML, but does not function as an LDAP provider. However, it can integrate with an LDAP provider, using the LDAP server to store and retrieve its own account data. When linked with an LDAP server, the combined system will allow for the same set of credentials to be used over both OAuth, SAML, and LDAP. Keycloak is difficult to use for the average self-hoster, both in installation and operation. Installation by itself is difficult, as Keycloak does not ship with a database driver; instead, it expects to operator to provide it with a database driver library and edit configuration to use this driver[28]. To the best of my ability, I have only been able to successfully install this driver given the exact version of the PostgreSQL driver used in the tutorial. Furthermore, integrating Keycloak with LDAP is not a simple task either; Keycloak expects the configuration data to be in a highly precise format, which the most common Linux LDAP server, OpenLDAP[22], does not support. This means that this integration requires editing significant amounts of OpenLDAP

configuration, which is rather complicated. Operation of keycloak is not trivial, as it has many features that are not required for domestic self-hosting, such as the addition of multiple realms (i.e. complete separation of identity sets), complex access control, complex management of connected apps, etc. Furthermore, managing apps that connect through LDAP does not go over Keycloak at all, further complicating matters.

Authentik[6] as an open source identity provider written in Python. It natively supports OAuth, SAML, and LDAP, and is focused on flexibility and versatility. It supports many additional features, such as user enrollment, conditional access, and federation. While these are not bad qualities by themselves, they do cause Authentik to become overwhelmingly feature-rich and complex for domestic use. Furthermore, Authentik can only be installed via Docker, making it unsuitable for the approximately 29% of domestic systems that do not use Docker. Installing Authentik on the approximately 58% of systems that only have part of their services dockerised might also be difficult, as it is not always trivial to have a non-dockerised service communicate with a dockerised service.

3 Related Work

Wong[62] considers the use of self-hosting in contrast to vendor/consortium based hosting by galleries, libraries, archives, and museums. Data was collected via a survey with 21 participants active in some position in one of these institutions. The amount of institutions surveyed that used self-hosted software was approximately equal to the amount that did not. The study found that the choice whether to self-host is most often motivated by some sort of constraint, such as the available budget, the need for customisation, or the available in-house technical expertise. Despite this, most institutions do not regret their decision.

Molnar and Schecter[38] explores security issues that are new or greatly exacerbated when moving from self-hosting to cloud-based hosting, as well as their countermeasures. The study finds that most novel issues can be mitigated using various types of auditing, both technical and legal. It also notes that cloud hosting makes it economical to take certain security measures that scale well across a large number of machines, but would be too costly for smaller deployments. While the study is based around large institutions, the list of security issues does provide some background as to the benefits of self-hosting domestically, especially given that many of the proposed solutions to security issues with cloud-based services require expertise and/or effort by the party that uses the services that would not be feasible for a domestic user.

Angeli et al.[3] criticises the growing use of cloud based software in higher education, citing worries over the transfer of political power to a small set of companies, as well as privacy, worker's rights, and environmental concerns. The study also proposes policies that could reverse this trend and provide additional benefits to the institution's community, and a concrete method for universities to return to self-hosting in a sustainable manner. Studying self-hosting in higher education comes from a different point of view than self-hosting for domestic use, as universities have historically self-hosted, and as such self-hosting in this context has already proven a practically viable concept. Nevertheless, many of the issues with cloud based software and some of the benefits of self-hosting cited in the study do also apply to domestic use, although the proposed method of self-hosting is focussed to too large of a scale to be relevant to this paper.

Mengyi et al.[30] proposes an augmentation of Shibboleth, a SAML authentication provider, adding support for OpenID Connect and theoretical support for any other authentication protocol. It does so not by changing the authentication provider itself, but by creating a separate component to interoperate between SAML and other protocols.

4 Protocols

This section will describe the different authentication protocols that shall be supported, by looking at both their technical documentation as well as some implementations. It shall then identify common features, and compare the protocols based on these common features. This section is intended to answer research questions 1.

4.1 Oauth

This section will describe a high-level overview of OAuth 2.0, and roughly describes the way it should be implemented in an authentication service for domestic self-hosting. For more information on OAuth 2.0, including implementation details, I recommend [44], which is where most information in this section comes from.

When mentioning OAuth, in both general usecases and throughout this document, one usually refers to OAuth 2.0. OAuth 1 did exist and was mostly used by large providers, but was found to (amongst other flaws) be unsuitable for web applications, and as such has been replaced by OAuth 2.0. For reasons of brevity and readibility, any mentions of Oauth in this document refer to OAuth 2.0 unless otherwise specified.

Due to reasons related to its development, there is no concrete specification for OAuth 2.0. Rather, there is a collection of separate RFCs[41], which don't necessarily cover all implementation details. Regardless, it seems almost all implementations have converged to a compatible protocol.

It should be noted that while OAuth can be used as an authentication protocol, it is not strictly intended as such. Instead, it was originally intended as an *authorization* protocol, i.e. a protocol that allows applications access rights to other applications; for example, to allow a third-party twitter client access to your Twitter account, or an automatic mail program to gain access to your inbox. This is a flow that is different from a dedicated authentication provider. This section will also explain the way in which a dedicated authentication provider and a client application that only connects to an OAuth server for authentication interact, and how this is different from other flows.

4.1.1 *Client Application Registration*

For a client application to connect to the OAuth service, it must first be registered with the OAuth provider. This

will generate a Client ID, and optionally a Client Secret, a confidential string which the application uses to prove its identity to the server. OAuth authentiaction involves redirecting the browser from the client application to the authentication provider, and then back to the client application; the URIs belonging to the client application to which the user should be redirected in this second step should also be registered with the authentication provider, to prevent a malicious party from impersonating a client application. Many providers also ask for additional information about the connecting application, such as the name of the application or an icon, which can be used to identify the connecting application to the end user when the application needs to be authenticated, or in the menu of the OAuth provider.

4.1.2 Basic Authentication Flow

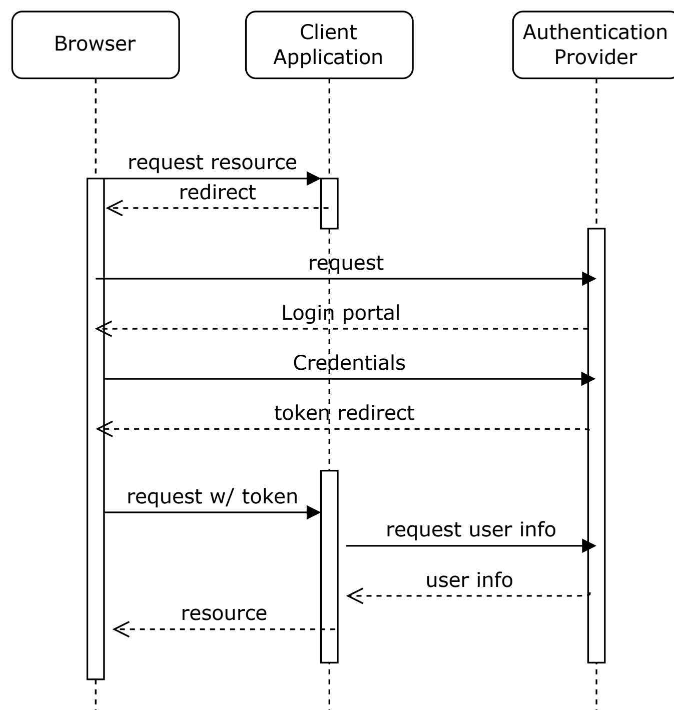

Fig. 1. A schematic overview of the basic OAuth flow

Once the application has been registered with the provider, end users can be authenticated to the application. This process is illustrated in figure 1 First, the end user accesses the login page from the client application. If the client application notices that the end user is not yet authenticated, it will redirect the user to the authentication provider, using a URI that contains several GET parameters, most notably the applications Client ID. The authentication provider then verifies the identity of the end user, completely independent from the client application. The authentication provider can use any process for this verification. Once the identify of the end user has been verified, the authentication provider redirects the user back to the redirect URI that was registered by the application, and appends a so-called code to this URI as a GET parameter.

At this point the client application still knows nothing; not only does it not have any information regarding the identity of the end user, but it also has not yet verified that the code it was given by the user's browser actually originated from the authentication provider. In order to verify this, it must exchange the code for an access token. The request to do this will contain the code, the client id, and the client secret. This will return an access token, which must be included in any requests to the authentication providers main API, in order to verify the identity of the client application and the user for which it is trying to access information to the authentication provider, as well as authorize the client application for this particular user. It should be noted that the code can only be used in a short window, and only once; if a code is ever used twice, the authentication provider should assume it has been compromised, and invalidate not only the code itself but also any access tokens that have been generated as a result of it.

At this point, the client application knows that the user is logged in, but it does not know any information about the user, and can therefore not distinguish this user from any other user it might know. To get information on the user, it can query the authentication provider, which should provide an API endpoint that returns some data on the user; of course, querying this endpoint requires an access token. How exactly the user information is retrieved will be described under section 4.1.5

4.1.3 Variations

Applications that execute all code on the end users device do not have a way to hide their client secret from the public. To accommodate this, they may omit the client secret when exchanging their code for an access token. Using this authentication flow, there is no for the authentication provider to verify the identity of the client application; the only way to make this flow secure is to ensure that the code can only be received by the client application, which is done by registering the redirect URL with the authentication provider, using the state parameter, and only allowing secure connections.

There is also the even simpler implicit flow, where the redirect URL does not contain a code, but rather an access token; this flow does not require the client application to make any requests to the authentication provider to receive an access token. While this was quite popular for webapps in the past, some OAuth provider implementations no longer support this flow due to security concerns.

Security can be enhanced by using the Proof Key of Code Exchange, or PKCE addition. Here, the client application generates a random string of characters, called the code verifier, and the hash of this string, called the code challenge. The challenge is added to the parameters when initially redirecting the user to the authentication provider. The verifier is added to the parameters when exchanging the code for an access token. This allows the server to verify that the request for the access token comes from the same program that redirected the user to the authentication flow that generated the corresponding code. Not all authentication providers support this addition for all authentication flows, and some don't support it at all. This addition is mainly recommended to be used for flows without a client secret, but specifications allow this addition for all authentication flows (except implicit). It is recommended for client applications to use PKCE whenever it is supported.

4.1.4 Notable details

The above explanations repeatedly mention HTTP related terms, such as "redirect" and "GET parameters". This is because OAuth is exclusively an HTTP-based protocol, and requires the end user to be able to interact with a browser. While this is not a problem for web applications, it does mean that desktop or mobile applications will need to either ship a browser with the application, or integrate with the system browser.

The communication between the end users browser and the authentication service, as well as the redirect that occurs immediately afterwards contain sensitive credentials, that could be used to hijack the session or even the whole application. As such, these requests must be handled over a secure connection, and may not be encapsulated in an iFrame.

Any URI that redirects the user towards the authentication provider can also include a state parameter. The contents of this state parameter will then be included when the user is redirected back to the client application. The client application can use this feature in two ways. Firstly, it can be used it to store information related to the users original request, such as a location the user should be redirected to after authentication has succeeded. Secondly, the application can use it to verify that a request which contains a code corresponds with a user requesting a login, which can help prevent CSRF attacks.

The request which initially redirects the user to the authentication provider can include a list of scopes. This can allow for fine-grained control over what the resulting access token can be used for.

4.1.5 OpenID Connect

As mentioned previously, OAuth is an authorization protocol, but not necessarily an authentication protocol. While it does allow the end user to log into an application via a centralised authentication flow, it does not provide the client application with any information about the end user. This information would have to be retrieved separately using the provided access token. As OAuth does not specify the method by which this information should be retrieved, the format in which this information should be presented, or even which information should be included in such a request, it cannot be seen as a standardised authentication protocol.

OpenID Connect is an authentication protocol built on top of OAuth[53]. It defines a several ways to use OAuth to provide authentication services, including all required implementation details, and several extensions to the OAuth protocol. Firstly, the OpenID Connect specification defines a standardised set claims that an authentication provider can make, as well as formats in which these claims can be communicated. It also defines a set of scopes under which various claims fall; for example, an end users name is under the basic scope `profile`, whereas their email address is under the `email` scope. Finally, it defines two ways to retrieve this information.

The simplest way does not require any OAuth extensions. In this flow, the client application first obtains an access token through the normal flow. It can then use this access token to access the `userinfo` endpoint, which will return information about the end user which granted the access token. The client application can trust this information, as it has been retrieved directly from the authentication provider. The main advantage of this method is the conceptual simplicity; once one has a basic understanding of the OAuth process, it becomes trivial to understand not only how this flow works, but also why it can function as a secure authentication mechanism. This in turn makes it easy for both the authentication provider and the client application to properly implement. The disadvantage of this method is in its inefficiency: after the user has been redirected to the client application with an access code, the client application first needs to exchange this access code for an access token, and then retrieve user information with the access token; so a client application has to make two successive requests to the authentication provider for each login.

To improve efficiency, OpenID Connect defines an extension to the OAuth protocol, called an ID token. This requires the authentication server to have generated an asymmetric key pair, and publish the public key. In places where normally an access token is send, an OpenID Connect server (additionally) sends over an ID token, which contains user information, and is signed with the generated private key. This means the client application can verify that the data is valid without having to contact the authentication provider. This ID token can be obtained through two flows: the "code" flow, and the "id_token" flow.

The "code" flow is entirely equivalent to, and must always be combined with, the standard OAuth "code" flow. The only addition is that when the code in exchanged for an access token, the server sends an ID token alongside the access token. This maintains all the security aspects of the flow using the `userinfo` endpoint, but has one fewer API call.

The "id_token" flow is equivalent to the OAuth implicit flow. Here, the authentication provider sends an ID token to the client application when the browser is redirected to the authentication provider. In this flow, the client application does not need to directly contact the authentication provider at all.

The fully implicit flow does raise some security concerns; if the redirect is tampered with, potentially sensitive user information could leak. This can be mitigated by combining the two access flows: OpenID Connect allows for a client application to request both a code and an ID token in the same request. In many of these cases, the ID token only contains the unique user identifier; this means the client application can distinguish the user and perform actions that don't rely on (up-to-date) user information without making an extra call to the token endpoint, while still locking the potentially sensitive user information behind a proper OAuth procedure.

A client application appropriately using OpenID Connect must know quite a lot of things about the authentication provider before starting the authentication process. For example, it must know the exact URIs for several endpoints, and it must be aware of the public keys the authentication provider is using to sign its data. To ensure that this information does not all need to be provided to the client application by hand, there is a standard that specifies which

information an authentication provider should publish, and how[54].

An authentication server must publish a document on a URL that is fixed relative to the URL it sets as an issuer value. This document must include the issuer name, location of the authorization and token endpoints, the location where the public keys the authentication provider uses for signing ID tokens can be found, and the supported flows and cryptographic algorithms. It is recommended to also include the location of the `userinfo` endpoint and the supported scopes.

4.1.6 Considerations for DAS

The specification for OAuth, and to a lesser extent that of OpenID Connect, leave quite some details open to the implementer. Most notably, there are a variety of different mechanisms and features that an implementer can decide to support or not support. On the one hand, the proposed service should support as much as possible, to maximise compatibility. On the other hand, some mechanisms are significantly less secure than other mechanisms, especially when configured improperly.

I see relatively little value in fully supporting discovery; there are very little self-hosted services (or any services at all for that matter) that use discovery of OpenID Connect servers, and correctly implementing it would be quite a lot of effort. Some parts of the discovery standard however, most notably the provider configuration request, should be supported, as some self-hosted applications, e.g. Gitea[23] use this feature to automatically configure themselves as client applications. Dynamic client registration should not be supported, as the server administrator should be fully in control of which clients can connect to the proposed service.

The default ID Token signing algorithm is RS256. This algorithm should be supported by any client and is reasonably secure, and as such I can see of no reason to support any other algorithm.

Token ID and userinfo encryption is rarely implemented by self-hosting applications, and provides little security benefit when the connection itself is encrypted using TLS, and even less benefit if the connection is between two services on the same machine, as is common in domestic self-hosting. However, this feature does require a significant amount of effort to implement. As such, this feature should have the lowest amount of priority for the system.

Backchannel and frontchannel logout are not required for the basic authentication flow to function. As such, they should be considered a relatively low priority.

The implicit and ID Token flows can pose a security risk if the application is not properly configured. As such, precautions should be taken to ensure that these flows are configured properly. Notably, it should be strictly enforced that the redirect URLs strictly match those that are configured, and that all traffic occurs over HTTPS. The possibility of providing only a limited amount of information via the ID Token, and locking more sensitive information behind the userinfo endpoint should also be considered.

The implicit flow can be considered a security risk; it should be investigated to what extent compatibility will be compromised if PKCE is mandatory for this flow.

If not mandatory, claims and scopes supported should be limited to those that have an equivalent in other authentication protocols, to maintain a consistent experience for the end-user.

Support for different locales can be a very valuable feature for some user groups, but can also be difficult to implement, and is not required for base functionality. Furthermore, this feature is not present in other authentication protocols, and may greatly complicate using the service for the administrator. This feature is not necessary for a minimum viable product; whether or not this feature should be supported after this stage requires further market research.

Support for the claims and request parameters is not required for a minimum viable product, as any client application should be able to use an authentication provider that does not support these features. To account for the possibility that a client incorrectly assumes that this feature is supported, however, adding support for this feature should be considered at a relatively low priority. The low priority is because I deem it relatively unlikely any client application will break if this feature is not supported, whereas this feature is relatively complex to implement.

Apart from what is listed above, all optional features should be implemented. Most notable features that should be implemented are PKCE support, support for both public and pairwise subject types, and support for all grant types described in [53] (with the aforementioned caveats for the id token and implicit flows). However, it should be noted that these features are optional, and may have to be dropped due to time constraints.

4.2 LDAP

The Lightweight Directory Access Protocol (LDAP) is a directory database which is commonly used to store user information. It is designed merely as a database, but a specific procedure allows client applications to verify user credentials against the database. In this procedure, unlike in OAuth, the client application is still responsible for gathering the credentials. It can then exchange these user credentials for authenticated information about the corresponding user.

4.2.1 Database Structure

Unlike more traditional object-relational databases, a directory database contains a tree-like structure, where each entry holds data of its own, but is also a parent entity to other entries (which may in turn be a parent to more entries). Entries hold data in several fields; each field has a name, and holds one type of data. Entries can be constrained by a class, which forms a template for which fields an entry must have and which optional field it may also have. All entities in LDAP have a unique Object IDentifier (OID). Furthermore, entries can be uniquely identified by their Distinguishing Name (DN), which is a list of attributes which can distinguish entries within the children of their parent, similar to a filepath[65].

In order to use LDAP for authentication, there must be some sort of entry class that represents a user for authentication[33]. This object class must have at least two fields:

firstly, some sort of unique identifier that is memorable to humans, such as a username or an email address, so that the client application can connect the entry to the user credentials. Secondly, a password field, so that the LDAP server can authenticate for this entity. This password field does not have to be in plaintext; in fact, it is now generally recommended that this password field is a hash[64].

4.2.2 Client Application Registration and Configuration

Whether or not the client application needs to be registered with the authentication provider depends on how the authentication provider is configured. It is possible to connect to an LDAP server with or without client application credentials, and an LDAP server could be set up such that unauthenticated client applications can authenticate users. For security reasons, however, it is advisable that a new set of client application credentials is generated for each client application, and that only authenticated client applications can access any user data. As such, the authentication provider should be configured to generate a new set of credentials for use by the client application, and the client application should be configured to use those credentials.

The client application must be configured in order to be able to match the credentials provided by the end user entries in the LDAP database, and in order to retrieve information about the end user from the database. Specifically, it needs to know how to identify and search for entries that represent users, and which fields of these entries contains what information about the end user.

4.2.3 Authentication Flow

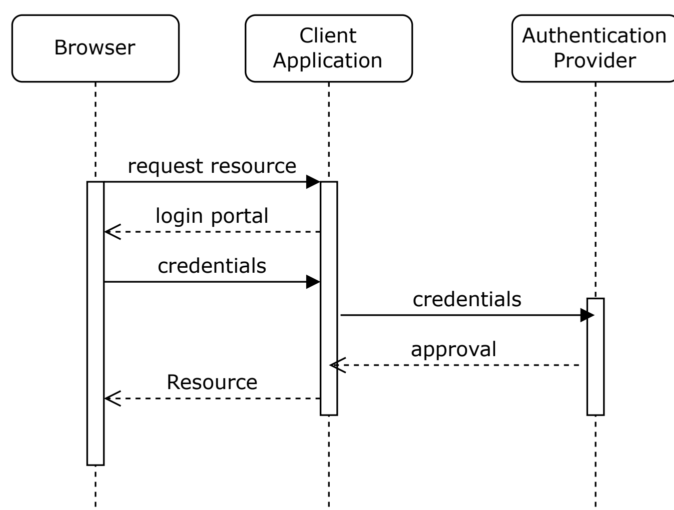

Fig. 2. A schematic overview of the events in LDAP authentication

The sequence of events required for authentication can be seen in figure 2. Firstly, the client application collect the user credentials, after which it connects to the authentication provider. If applicable, the client application authenticates itself with the authentication provider immediately after connecting, in order to be allowed to query the database. It then searches for user entries that correspond to the identifying part of the credentials. If it finds any, it will extract the DN of the user entry.

The combination of the DN of a user entry and its password field serve as credentials to authenticate with the authentication provider. Once the client application has retrieved the DN of the user entry, it will attempt to authenticate itself to the authentication provider with this DN and the password provided by the user. If this authentication succeeds, the client application knows the user information corresponding to the credentials provided by the end user, and it knows that the credentials are correct; the end user has been identified and authenticated.

4.2.4 Considerations for DAS

The proposed service does not need to feature as a full LDAP server; it only needs to provide read access to data with a predefined structure. Furthermore, given the domestic use, there is no need for any complex access control. This means all entries can be of the same class, and reside under the same parent entry.

As such, only a limited subset of LDAP protocol functionality actually needs to be supported. Any operation that modifies the data should not be supported, as modifying the user data should only be possible for administrative end users. Looking strictly at the authentication flow, only the Bind, Search, and Unbind operations are strictly required; however, the Compare and Abandon operations also do not lead to any modification of the data, and should be supported to maximise compatibility. The most notable distinction between the requirements for the proposed service and an actual LDAP server, however, is that the information that should be searchable through this LDAP database does not actually need to have a nested directory structure, but merely needs to be presented as if it does.

4.3 Forward Authentication

The forward authentication protocol relies on a reverse-proxy to verify with the authentication provider that a request has been authenticated before forwarding the request to the client application. When proxying requests to the client application, the reverse-proxy sets headers that the client application can use to identify the user.

Note that all the client application needs to do is to read the user information from the correct HTTP headers it receives; it does not need to make any requests or maintain any state. Even if the client application does not support this authentication protocol, this would only mean it is unable to identify the user; the user would still have to be authenticated before they are able to make any requests. As such, this authentication protocol can be used to ensure unauthenticated users can't make requests to applications that otherwise don't offer authentication, although the application would be unable to distinguish users.

There have been a variety of applications that provide functionality for forward authentication, with Authelia[37] being the most prominent modern example in domestic self-hosting[7].

4.3.1 Client Application Registration

For this authentication protocol to be secure, all requests to the client application must go through the reverse-proxy. This means that the client application must be listening

on an HTTP connection that is not publicly accessible. The reverse-proxy must be configured to listen on the desired publicly accessible connection, and know where to forward the requests to. The reverse-proxy must also know how to contact the authentication provider (see section 4.3.3).

If the client application supports this protocol, i.e. it can gain user information from incoming HTTP headers, it must know which headers correspond to which information about the user; this is usually a configuration option.

4.3.2 Basic Authentication Flow

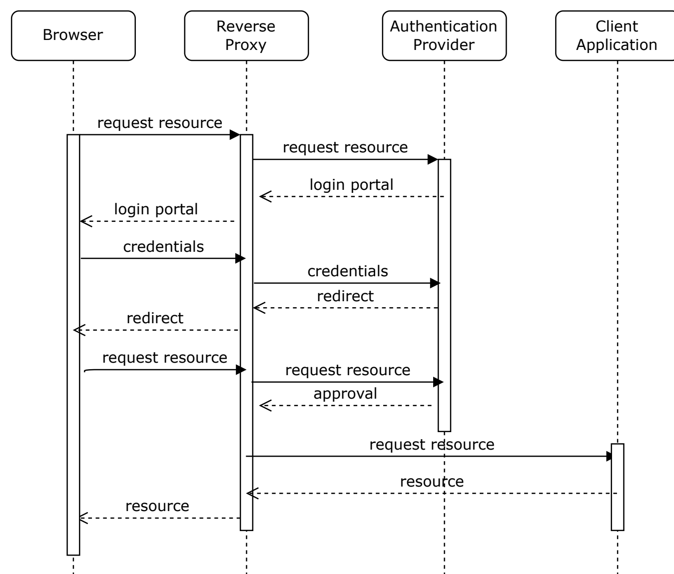

Fig. 3. The sequence of events in forward authentication

The flow for forward authentication is illustrated in figure 3. When the end-user's browser wants to make a request to the client application, the reverse-proxy first forwards the request to the authentication provider. On the initial request, the authentication provider will then notice that the user has not yet been authenticated, and communicate this to the reverse-proxy by returning an "unauthorized" status code. Instead of proxying the request to the client application, the reverse-proxy will instead return the login page to the end-user's browser, or a redirect to it. The end-user will then authenticate themselves to the authentication provider. Once this has been completed, the authentication provider will direct the end-user's browser to make the initial request again. Now, however, the authentication provider will detect that the end-user has been authenticated, and return an "ok" status code, as well as some headers that contain user information. The reverse-proxy will now proxy the original request, plus the added headers from the authentication provider, to the client application.

4.3.3 Integration with Reverse-Proxies

Forward authentication requires integration with a reverse-proxy, and the reverse-proxy must be configured quite specifically. It not only needs to be configured to first forward each request to the authentication provider, but it must also be configured to redirect to the correct login page in case of bad authentication, and to pass the correct headers from the authentication provider to the client application. It's especially important that the headers that the authentication provider uses to communicate user information will not be propagated from the incoming connection to the client application, otherwise a malicious actor could impersonate any user.

Not all reverse-proxies can be configured to use this authentication method. Most notably, Apache[21] lacks support for a module that can forward incoming requests to an authentication provider. In order to fully support such reverse-proxies, a technology was developed that serves the same purpose but in a different manner, called proxy authentication. This is further elaborated in section 4.4.

4.3.4 Notable Details

In this protocol, the client application will assume the information contained within the headers that identify the user, as it has no way to verify their authenticity. This makes it extremely important that the reverse-proxy is the only way to access this client application. If an attacker can access the client application while bypassing the reverse-proxy, they can set these headers to arbitrary values, which would allow them to impersonate any user. Similarly, the reverse-proxy must be sure to not proxy the headers that contain the user-identifying information. These potential security vulnerabilities make the system less secure than e.g. OAuth, where the client application must perform work to ensure that any information regarding the identity of the user comes from the authentication provider.

4.3.5 Considerations for DAS

How to deploy this protocol integrated with an external reverse-proxy is highly dependent on the reverse-proxy, can be rather complicated for the system operator, and is very hard to automate, as the authentication provider is not aware the client applications. This could be resolved by making the proposed service automatically write configuration for the reverse-proxy. This feature, however, is at the very edge of the scope of this project, and should be considered a wishlist feature. If it were implemented, it should at least include support for the Nginx auth request module[29].

It is also possible to delegate this protocol to an external provider. Authelia[37] functions as a reverse-proxy authenticator and supports authenticating its users over LDAP. Vouch Proxy[5] is a reverse-proxy authenticator for Nginx that supports authenticating its users over OAuth. Both of these external providers only support integrating with an external reverse-proxy, and as such don't work on systems that rely on Apache. Furthermore, these programs would have to be installed with the proposed service. While both options have sufficiently permissive licences to be shipped with the proposed service, integrating them might be more effort than including an implementation of this protocol in the service itself.

4.4 Proxy Authentication

Proxy authentication offers similar functionality to forward authentication, but does not require support from a pre-existing reverse-proxy. It does so by turning the authen-

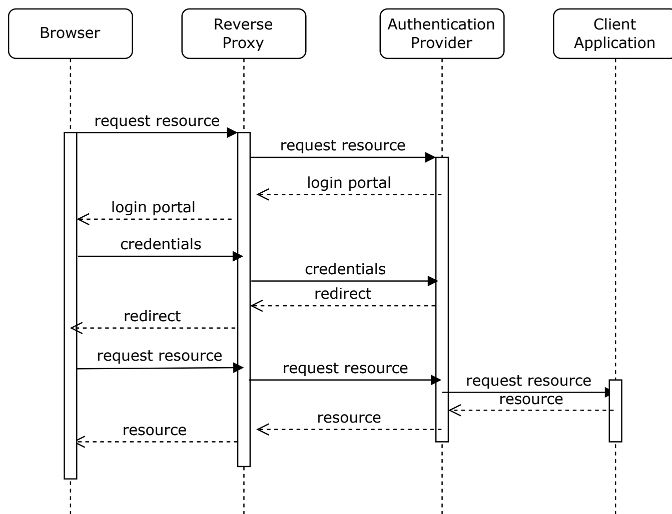

Fig. 4. The sequence for proxy authentication

tication provider into a reverse-proxy which only proxies authenticated requests.

4.4.1 Client Registration

Registering a client application for this protocol requires two pieces of information. Firstly, the authentication provider needs to know where to proxy authenticated requests to; in other words, the precise URL where the client application can be reached. Secondly, the authentication provider could be providing services for multiple client applications for this protocol, and needs some way to identify to which client application an incoming request should be proxied. This could be done in a variety of ways; for example, one could open a new listening socket for every client application (necessitating extra configuration for the previous layer reverse-proxy), or one could make the domain under which the client application is accessible a registration value, and use the `Host` header of each incoming request to dereference.

4.4.2 Authentication Flow

The authentication flow can be seen in figure 4. The ingress reverse-proxy, e.g. Apache or Nginx, never forwards a request to the client application. Instead, it proxies the request to a designated socket of the authentication provider. The authentication provider then determines if the request was made by an authenticated user, via e.g. the session cookie. If the user is not authenticated, the authentication provider returns some way for the user to authenticate, such as a login portal. After the end-user has authenticated themselves with the authentication provider, the authentication provider will redirect their browser back to the URL they originally requested. Once again, their browser will contact the reverse-proxy, which will once again first proxy it to the authentication provider. This time, however, the authentication provider will see that the end-user has been authenticated, and proxy the request further to the client application, with the addition of extra headers.

4.4.3 Considerations for DAS

This particular protocol requires DAS to actually proxy every HTTP request to the client application. As such, it is important that the technical aspects of this proxying function correctly and quickly. Most notably, it should correctly handle chunked encodings, and start sending data back to the end user as soon as its receives the data. In order to accomplish this, it may be necessary to skip any HTTP libraries and operate directly on TCP level, as only a very limited interaction with HTTP features is required for the proxy to correctly function.

4.5 SAML

The Security Assertion Markup Language (SAML) is a protocol that allows one system to communicate to another system that it has verified some facts about an entity[8]. It can be used as an authentication protocol, an authorization protocol, or as a protocol to communicate that some arbitrary data has been verified. When used as an authentication protocol, it allows the authentication provider to communicate an assertion to the client application about the identity of the end-user.

SAML has been on version 2.0 since 2005. As such, this document assumes older versions have by now been phased out. Any mention of SAML is this document will exclusively refer to SAML 2.0 unless explicitly stated otherwise.

SAML is quite extensible. The core standard [8] only specifies a very basic description of the parties that can be involved, and the format of messages that can be exchanged independently of how these messages are exchanged or what information they convey. The extensibility of the standard means that two applications which support SAML may not necessarily be interoperable, as support for SAML does not imply a shared set of supported communication protocols. The SAML committee has defined how certain communication protocols should be used [9], but explicitly allows the creation of new standards for different protocols, as well as new standards that use a protocol differently from existing standards using the same protocol.

In order to ensure interoperability between different SAML implementations, several ways to use various messages and protocols defined in [8] and [9] have been defined, which are called profiles [11]. The only purpose of DAS is to be an authentication provider, and as such only the SSO profiles are of interest for this document. Since most self-hosted applications are web-based, the Web Browser SSO Profile is of greatest importance for this document.

4.5.1 Client Application Registration

The SAML committee has defined a metadata standard [10], which specifies a single XML document that contains all information that is required for a client application to interact with an authentication provider, or the other way around. This document specifies which data the relevant party wants to exchange using SAML, and which role it will play in this exchange. It also specifies the public keys the party will use, the location of certain endpoints, and the support or preference for certain optional features.

In order to register a client application with the authentication provider, the authentication provider needs to be aware of (where it can find) the metadata for the client application. In order to configure the client application to use the authentication provider, the client application needs

to be aware of (where it can find) the metadata for the authentication provider.

4.5.2 Methods of data exchange

The main SAML specification [8] intentionally does not specify the means by which data is exchanged. The profile specification [11], which specifies which messages are to be exchanged to log in via SSO, specifies that three different ways of communication have to be supported for authentication via the browser; these methods themselves are specified in the bindings specification [9]. As each of these three methods can be used at multiple points in the authentication flow, this section will briefly describe these communication methods before the authentication flow itself is described.

HTTP Redirect Binding: In this method, the initiating party redirects the user's browser to a designated endpoint at the receiving party. The SAML message is encoded in base64 and appended to the redirect URL as a query parameter.

HTTP POST Binding: In this method, the initiating party generates an HTML form, the target of which is a designated endpoint at the receiving party. The SAML message is encoded in base64, and placed inside the form as a hidden element. The only other element of the form is the submit button. The SAML specification suggests using JavaScript to automatically submit the form.

HTTP Artifact Binding: In this method, the initiating party uses either of the above methods to transfer a much shorter message to the receiving party. This message contains only a small reference text, known as an artifact. The receiving party must then contact the initiating party directly via a different means, where it can exchange the artifact for the full contents of the message. It may be required to use cryptographic signing to authenticate itself in this request.

It should be noted that all three communication methods allow for an additional parameter called `RelayState`, which can be included in the request, and should be communicated unaltered to the response.

4.5.3 Authentication Flow

When the client application determines that a user is not authenticated, it uses any of the above three methods to redirect the end-user's browser to the authentication provider. In doing so, it also gives the authentication provider a corresponding SAML message. This message identifies the client application that made the request to the authentication provider, possibly with a cryptographic signature. The authentication provider then authenticates the end user.

Once the end user has been authenticated, the authentication provider uses either the HTTP POST Binding or the HTTP Artifact Binding to redirect the user back to a designated endpoint at the client application. Note that the HTTP Redirect Binding can not be used in this step; this is because the SAML message that is communicated in this step is assumed to be too large to fit inside a URL parameter. The associated SAML Message contains sufficient information for the client application to identify the user. What information it contains exactly is dependent on the authentication provider, as well as the preferences indicated by the client application.

4.5.4 Comparison with OAuth

The authentication flows in SAML and OAuth are quite similar. This section will first go over the "code" and "id_token" flows of OpenID Connect and analogous flows in SAML, and then describe some notable differences.

Both OAuth flows first communicate the request for authentication from the client application to the authentication provider by including the request data as a URL parameter in a redirect. This is analogous to initiating a SAML request via the HTTP Redirect Binding. Afterwards, the "code" flow returns a code via another HTTP redirect URL, which the client application can exchange for an ID token with a request directly to the authentication provider. In SAML, a similar situation occurs if the authentication provider opts to respond with an artifact. Alternatively, the "id_token" flow returns some signed information about the user in a redirect. While actually returning this information in a redirect is not possible under SAML, using the HTTP POST Binding for this is equal in all aspects except for the HTTP method used and the way it is triggered.

Of course, there are also some very notable differences between SAML and OAuth. One of the most immediately noticeable differences between the two protocols is that SAML is based on XML, in contrast to OAuth's JSON. This also means that SAML is much more formalised, which each element of a SAML message explicitly referring to a standardised format, and as a result much more verbose. For example, what might in Oauth simply be represented by `'FirstName': 'Steven'`, is represented by text that can be seen in figure 5. This also helps to explain why a SAML message that contains user information cannot fit inside a URL.

The existence of the POST binding is also significant.' In OAuth, all requests use GET requests, either via the user's browser and a redirect or between the client application and the authentication provider directly; in SAML, either party can also use a POST request via the user's browser to communicate data.

Another notable difference is that in SAML, the initial request from the client application to the authentication provider, which requests that the user be authenticated, can be passed by reference. In OAuth, this messages can only be passed via a URL parameter, whereas in SAML, it can also be passed via a reference which the authentication provider has to request directly from the client application. This is especially noteworthy since it is now the authentication provider that has to make a direct request to the client application, a situation which never occurs is OAuth.

It should also be noted that OAuth is entirely dependent on HTTP and web browsers to function as per its definition, whereas SAML also has many other definitions that allow it to function independently of these protocols. However, as the profile specification [11] doesn't specify any way to provide authentication services that is not dependent on a web browser, it is questionably to what extend any SAML-based flows that don't rely on web browsers are in use, and to what extend they are interoperable.

In general, it is a significant fact that while OAuth has a few strictly defined flows, SAML has many layers of different specifications, all of which allow for some choice on the part of the implementers of both parties. This also

```
<saml:Attribute
xmlns:x500="urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
Name="urn:oid:2.5.4.42" FriendlyName="givenName" x500:Encoding="LDAP">
<saml:AttributeValue xsi:type="xsd:string">Steven</saml:AttributeValue>
</saml:Attribute>
```

Fig. 5. How SAML communicates a given name[13]

has results for interoperability; in OAuth, any pair of client application and authentication provider will function as long as they implement the core standards, with the only possible failure occurring if either party does not think the other meets their security standards. In SAML, on the other hand, both parties can implement only a subset of the available options, and interoperability is dependent on an overlap of these subsets; it is entirely possible that two parties which both correctly implement the SAML protocol are unable to communicate with each other.

4.5.5 Considerations for DAS

SAML is supported by a relatively small amount of applications used in domestic self-hosting, and all popular applications that do support it also support OAuth. As such, implementing support for SAML is not a high priority. However, if implementation can be done sufficiently easily, it could be a nice additional feature.

As noted in section 4.5.4, there are several flows in SAML which are analogous to flows in OAuth. In fact, for these particular flows I can see no differences in the actual logic, only in the syntax of the messages and HTTP responses. As such, it should be possible to develop a SAML integration by using the same underlying logic as for OAuth, but with a different presentation layer. This does exclude some functionality, such as the ability to encrypt messages, or the ability to resolve any requests that are passed via artifact. However, implementing such features is relatively complex given the complexity of SAML to begin with, and as such should not be considered any priority given their limited additional value. Any implementation of SAML should be considered best effort, and care should be taken not to spend too much time attempting to make the SAML implementation compatible with all theoretical client applications.

4.6 PAM

Pluggable Authentication Modules is a system used by many Unix-like systems that allows applications to authenticate users in a generic way. The authentication provider must provide a set of C-shared objects that implement a series of predefined functions, which are then called by the application. The return value of the functions will tell the application whether or not the user has been successfully authenticated[20].

4.6.1 Client Application Registration

The authentication provider does not need to be aware of the specific client application. However, the PAM system should be configured such that applications use the functions provided by the authentication provider.

A standardised configuration can be made for each application to call a specific set of functions by some specific set of authentication providers. This configuration file specifies which modules to call, of what type they are, and how the result of each function call should be treated. For example, a successful call to one authentication provider could be sufficient to consider the user authenticated, whereas a successful call to another could require calls to additional authentication providers.

There are four types of function calls that an authentication provider can provide:

1) `auth`: verify that the credentials for the user are correct
2) `account`: verify that the user is allowed to access the application; for example, expired accounts may have valid credentials, but are still not allowed to log in.
3) `session`: setup the environment for use by the user
4) `password`: change the user password

4.6.2 Basic Authentication flow

The precise authentication flow depends on the configuration for the client application. Some client applications may have different requirements, and some client applications may be configured to call multiple authentication providers. This section will quickly go over the most basic flow, where the client application simply wants to verify credentials at one authentication provider, similar to an application that uses LDAP.

The application calls the function provided by the application provider. This function will attempt to get the username and password (or other authentication token) from the parameters passed by the client application. If these are not present, it will call functions at the client application that will request these credentials. The function then checks if the credentials are correct using some unspecified manner, and returns the appropriate status code.

Afterwards, the application should also call the `account` function, which will in some unspecified manner verify that the account actually has permission to access the application.

4.6.3 Considerations for DAS

The actual PAM Modules should be considered a completely separate part from the main application. These simply C functions only need to pass on the credentials to the actual service, which can provide actual verification.

It is possible to use the existing LDAP port for this, which would save time in writing the application; in fact, there already exist a system for providing PAM modules via LDAP[15]. However, configuration for such a system is

relatively complex, as the LDAP system could be remote, and needs authentication credentials. [15] solves this by using a background service, but this service needs to be configured; for this usecase, the system would become too complex to justify using a PAM-LDAP binding as saving implementation time.

Instead, it seems to me the simplest option would be for the proposed service to create a Unix socket which is called by the PAM modules. This socket would be available to all services on the local system, but not to any outside parties. Since the only thing this socket needs to do is to verify given credentials, there are very little security risks associated with exposing this socket to the local system.

4.7 Comparison

This section will compare the protocols on a variety of features. Firstly, the features will be described. A comparison based on these features can be found in table 1.

Interface

Which party has control over the interface where the user logs in. This could be the client application, the authentication provider, or, in theory, some third party. In the table, "authentication provider" has been shortened to "provider", and "client application" has been shortened to "client".

This feature is important when it comes to the design of the proposed service, as it decides how much freedom the authentication provider has in how it determines the identity of the end user. If the authentication provider has interface control, it can use any number of methods to identify the user; if, however, it does not, it can only use the information that is passed onto it by the client application, which in the case of LDAP, is limited to two strings (a username and a password).

Communication

How the client application and the authentication provider communicate. This could be directly, via the users browser, via some third party, or a combination.

This feature outlines the major architectural differences between the protocols. In doing so, it also highlights the limitations of some protocols; for example, any protocol that communicates via the browser can only be used in contexts where there is a browser to speak of. It is also extremely important to consider when building an authentication provider, as it specifies what sort of communication methods the authentication provider must support

Data format

Which data format data is primarily exchanged in. This could be JSON, XML, some custom format, etc.

This feature mainly highlights the time in which a protocol was designed, as well as its intended audience. In particular, it highlights the difference between OAuth and SAML. It is also relevant for implementing an authentication provider, as the authentication provider must be able to parse the data formats for all the supported protocols.

Registration

Whether or not the authentication provider needs to be aware of a specific client application before it can be used.

This feature is reflective of the level of trust that needs to be established between the authentication provider and the client application in order for the protocol to function; if a protocol requires registration of the client application, this means that the authentication provider will only provide services to clients that it already trusts.

This feature must also be considered for some additional functionality. An authentication provider could offer to only provide authentication for certain applications for some users; however, this is only possible if the authentication provider can distinguish which client application an authentication request comes from. If the protocol does not require registration, this may not be possible.

Awareness

Whether or not the client application needs to be aware of the authentication provider it is using.

This feature highlights the unique benefit of the reverse proxy setup, which is that the client application does not need to be aware of the authentication provider. This allows the protocol to provide authentication services for a client application that would not otherwise support a centralised source of authentication.

Range

The "furthest" type of connectivity that can be used to communicate between the authentication provider and the client application. The two applications could be on the same machine, but they could also be separated by an internet connection.

This feature shows the limitations of protocols that do not require registration; table 1 shows that these protocols have only a limited range. It is also relevant to consider for implementing an authentication provider, as it shows the level of trust that one can expect from a client application; any requests coming in from the internet should be treated with more scrutiny than those coming from the local machine.

4.8 Authentication Schemes

Now that all protocols have been analysed and compared, one important question can be answered: what schemes are available to the authentication provider for authenticating the user?

For OAuth, SAML, and Reverse Proxy, the authentication provider needs to provide an HTTP server that authenticates the users browser. This means that any authentication scheme that works in a web browser can be used.

An LDAP client, however, will only make two requests to the authentication provider; one where it verifies that the user exists, and one where it asks the authentication provider whether the users password is correct. This flow itself was only intended for authentication via username/password, but this does not mean that this is the only way in which it could be used. The fact that the LDAP client asks the authentication provider to verify the password, instead of verifying the password itself, allows

PROTOCOL	Interface	Communication	Data Format	Registration	Awareness	Range
OAuth	Provider	Browser, optionally HTTP	JSON	Yes	Yes	Internet
LDAP	Client	TCP	Custom Binary	Recommended*	Yes	Internet
Forward Auth	Provider	HTTP Proxy	HTTP Headers	No	No	Local Network
Proxy Auth	Provider	HTTP Proxy	HTTP Headers	Yes	No	Local Network
SAML	Provider	Browser, optionally HTTP	XML	Yes	Yes	Internet
PAM	Client	C shared objects	Custom Binary	No	Yes	Local Machine

*In theory the LDAP server does not need to be aware of the client application, but in practice it is recommended that each client application has its own set of credentials, which is effectively a registration procedure.

TABLE 1
A comparison of the different communication protocols

the authentication provider to, in this stage, perform steps that are not quite the same as verifying a password. It should be noted that the string which is provided to the client application as a password does not necessarily have to be an actual password, but can instead be any string, as long as the authentication provider can use this string to make claims about whether the one who entered it corresponds to the provided username. For example, if the authentication provider knows a public key associated with the user, the user could also provide a digital signature of its username and the current time in the password field.

Of course, working with digital signatures is far too inconvenient for the average user. While software could be developed to automate such a task, ensuring that this software works on every platform a user may want to use is not feasible. Username/password combinations, of course, have the advantage of being feasible under all platforms, but they are associated with some security concerns. To mitigate these, two-factor authentication has been increasingly adopted. Some two-factor authentication systems could still be used over LDAP, either by appending some data to the password, or by performing an out-of-band verification while the authentication provider appears to be verifying the password. However, only TOTP[34] and predefined codes can be used without requiring the end-user to purchase additional hardware or requiring the development of additional client-side software. Since TOTP offers more security and more usability[51], the method of choice for a 2-factor authentication system over LDAP should be to have the user append a 2-factor authentication code to the password entered in the password field.

For PAM, a variety of authentication schemes can be used in theory, although in practice it is questionable how many are supported by the client application; for maximum compatibility, the same constraints as LDAP should be assumed.

5 Architecture

This section will explain and motivate the most important high-level implementation decisions, as well as provide a solution direction to the most significant problems in implementing DAS. It will start by listing the requirements, and continue by providing a schematic overview of the proposed software implementation.

This section is written along the guidelines of ISO 42010, but provides no guarantees as to its precise compliance to this standard.

5.1 Stakeholders

This section will go over the four stakeholders that have been identified with brief descriptions of the stakeholders and the requirements they have of DAS. For each stakeholder, this section will also list some requirements. These requirements are split into hard requirements, which can be objectively measured, and soft requirements, which cannot. Furthermore, requirements can me mandatory, as indicated by the word must, meaning the service is only of use to this stakeholder if this requirement is met, or optional, indicated by the word should, indicating the service would be improved if this requirement is met.

5.1.1 End Users

This is the group of people who want to use client applications which rely on the service to perform authentication. They are mainly interested in the core functionality and security of the service.

Hard Requirements:

- The service must be able to serve as an authentication provider for OAuth
- The service must be able to serve as an authentication provider for LDAP
- The service must be able to serve as an authentication provider for reverse-proxy type flows (i.e. forward auth, proxy auth).
- The service must only authenticate users with the correct credentials
- The service must not allow any party other than the specific end user to obtain a copy of the end user's credentials
- The service must be able to function on a Raspberry Pi, with 90% of calls to the service taking less than 10ms, and without significantly affecting other services on the same system
- User credentials must not fall in the hands of an attacker, even in the case of a data breach
- The service should be able to serve as an authentication provider for SAML
- The service should be able to serve as an authentication provider for PAM

Soft Requirements:

- The service must be available to provide authentication services under all reasonable circumstances

5.1.2 System Administrators:

These are the people who administer the servers, and are responsible for setting up, maintaining, and administering the service. Note that, in domestic self-hosting, this is a subset of the end users. This group is additionally interested in the ease of setup, maintenance, and administration of the service.

Hard Requirements:

- The service must run on Unix-based operating systems, most notably Debian
- The shipped service should be less than 500MB

Soft Requirements:

- It must be easy to deploy the service for a person with limited knowledge on server administration
- It must be easy to administer the service for a person with limited knowledge on server administration and security
- The service must be compatible with the greatest number of other protocols.

5.1.3 System Integrators

This group is creating systems to automate domestic self-hosting, and would like to integrate the service as part of these systems. While they do want their systems to satisfy the needs of their own users, and as such share many of the requirements above, they also have a few unique requirements of their own, related to their unique way of interacting with the service.

It should be noted that since integration with systems that aim to automate domestic self-hosting is, in itself, considered optional, any mandatory requirements in this section are actually not mandatory for a minimum viable product.

Hard Requirements

- It must be possible to add, remove, and edit client applications through an interface suited for programmatic control.
- It must be possible to retrieve information on client applications through an interface suitable for programmatic control.

5.1.4 Developer

This stakeholder is responsible for the development of the service, i.e. the author of this paper. Their concerns are related to the ease and feasibility of the development process.

Hard Requirements

- The service must be written in a language the developer is familiar with
- Implementing the service must be possible within a timespan of 3 months
- Implementing the service should be possible within a timespan of 6 weeks

5.2 Position View

This section describes the position that DAS takes in a domestic self-hosting environment. It is intended for all stakeholders: for users, it shows what they can expect from the service, and where their data is stored; for administrators and system integrators, it additionally shows where the service fits in their system; for developers, it shows under what conditions the service is operating; and to all parties, it gives a rough idea of what the purpose of the service is.

The diagram illustrates how data flows between the different components that interact when a user is being authenticated; that is, the browser (or other client), the client application, and the authentication provider. The components themselves are indicated by rectangles with rounded edges, and their interaction by a bidirectional arrow. Arrows that indicated the transmission of credentials have been coloured red. The components have been coloured according to which protocol they are associated with; components that are using OAuth are coloured blue, components using LDAP are coloured orange, components using forward authentication are coloured green, and components using proxy authentication are coloured purple; the service itself is coloured white. Note that the position of any SAML components would be the same as OAuth components, and the position of any PAM components would be the same as LDAP components.

The diagram can be found in figure 6. The LDAP flow is the simplest, as the credentials are simply passed from the browser through the client application to the authentication provider. The OAuth flow is notably different, because the credentials are only exchanged from the browser directly to the authentication provider, with both the browser and the authentication provider exchanging additional data with the client application. The forward authentication flow is unique in that there is a component, the reverse-proxy, that performs more than one step in the authentication sequence; it makes a request to DAS, and then makes a request to the client application. The proxy authentication flow is unique in that DAS is not an final destination for all data, but a middleman that forwards data to and from the client application.

5.3 Choice of Language

This section will briefly go over all programming languages that the author is familiar with, and compare their advantages and disadvantages with respect to implementing the service, in order to finally arrive at a motivated conclusion for which language the service will be implemented in.

5.3.1 PHP

PHP[45] is a scripting language for the purpose of generating web pages on the server side. The PHP code integrates within an HTML document. In the standard configuration, the interpreter will interpret the page when it is requested; this means that once the reverse-proxy and PHP interpreter are properly configured, PHP software can be deployed by simply placing the relevant files in a directory where web pages are served from. This also means that each time the page is requested, the script is executed in a vacuum; different script execution calls can only affect each other via some external system such as a database.

Advantages:

- Software can be very easily deployed on systems on which PHP is already installed, which is very common in domestic self-hosting

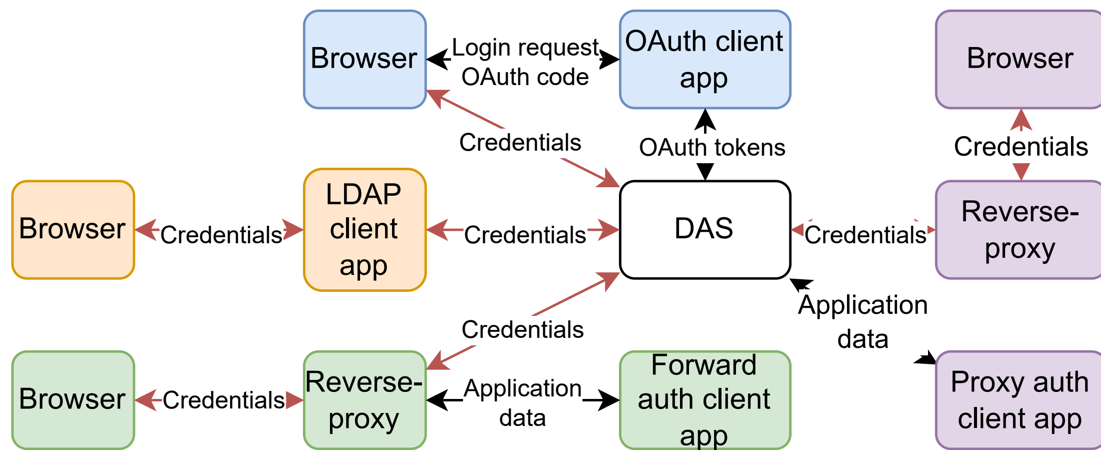

Fig. 6. How the service is positioned in a system

- Generating webpages is exceedingly simple
- Resource usage can be well constrained[57][55]

Disadvantages:

- Providing services for protocols other than HTTP can be difficult, as the language is built to be called whenever an HTTP request happens.
- Because all requests are isolated, it can be difficult to synchronise the different processes, which could make it more difficult to secure certain flows.

Verdict:
While the ease of deployment is a great advantage, the difficulty with implementing protocols that are not HTTP-based does not make it a good fit. Furthermore, the requirement for an external service to communicate between the different requests negates the ease of deployment, as this external service would have to be deployed and configured for this specific use. As such, PHP does not seem a good fit for this purpose.

5.3.2 Python

Python[50] is a general-purpose scripting language. It is commonly praised for its intuitive syntax and versatility.
Advantages:

- The ease and widespread use of the language could make the codebase more accessible, making it easier for third parties to contribute or make extensions
- The collection of available libraries can shorten development time

Disadvantages:

- Python is infamous for it's slow runtime, and the high CPU usage of its interpreter[35][48]
- Python services can be hard to deploy in web-based contexts, as the intended protocol UWSGI is not always well supported[4]

Verdict:
While the ease of code and commonality of Python are compelling, the interpreter is simply to resource-intensive and latency-inducing for this project.

5.3.3 Java

Java[14] is an object-oriented language, which compiles for a virtual machine. This virtual machine, known as the Java Runtime Environment, is available for a wide variety of operating systems; this means that the compiled software can be shipped to different operating systems.
Advantages:

- The fact that the compiled software can be shipped simplifies distribution to operating systems that would otherwise be rarely supported
- The language is well equipped with security features like encryption and data security[43]

Disadvantages:

- Java is notorious for its high memory usage[49][48]
- The Java Runtime Environment is not open source, which means it may not be available for all operating systems; most notably, some Linux distributions, such as Debian[60], don't provide the default runtime environment due to licensing reasons, but instead provide the open-source alternative OpenJDK[58]. This may lead to compatibility issues, especially when it comes to certain advanced features or libraries.

Verdict:
While the ease of distribution and security infrastructure would be convenient, the memory footprint as well as the potential compatibility issues make Java unsuitable for this project.

5.3.4 C++

C++ is a highly feature-rich programming language that is compiled to a native binary. Despite its age, it is still commonly seen as the tool of choice for applications where high performance is a must, such as drivers, reverse proxies, or video games. Programming in C++ is often said to be difficult, mostly due to the languages lack of garbage collection; instead, memory must be managed by the programmer, and failure to do so will cause the program to crash, possibly after exhausting the system memory.
Advantages:

- Resource usage and latency are extremely low [48]
- Compilers are available for almost every operating system
- Results in a single binary, simplifying installation
- There is a wide variety of libraries

Disadvantages:

- The lack of memory safety can, when not properly accounted for, result in unstable software, as well as several security vulnerabilities
- Libraries are usually linked, which means that the system on which the software is deployed needs to have the same libraries installed as the host system. Furthermore, this effectively requires a new compiled artefact for each operating system, as the libraries that need to be linked tend to be in different locations across operating systems.

Verdict:
The speed and richness in features and libraries make C++ a compelling option. However, the risk of security vulnerabilities due to memory mismanagement cannot be ignored.

5.3.5 *JavaScript*

JavaScript was originally designed as a language to run in the web browser, but has since been adapted to run as a back-end language by projects such as Node.js[40].

Advantages:

- Due to the fact that it is also used in web development, it is highly well known and therefore accessible to developers who may want to contribute to or extend the service
- The language has built-in facilities for HTTP calls

Disadvantages:

- A JavaScript server program requires a back-end to be installed
- The JavaScript package manager, npm[24], is notorious for hosting packages with an enormous amount of dependencies. This means simple projects can have hundreds of dependencies, which leads to large disk space usage. There have also been concerns that this may make programs vulnerable to supply chain attacks[42][31].
- JavaScript is notorious for liberally applying type conversions, which could become an issue when securely evaluating authentication data[47]

Verdict:
The installation difficulty, as well as the disk space usage, and potential vulnerabilities, make JavaScript an unsuitable language for this project.

5.3.6 *Elixir*

Elixir[59] is an evolution of Erlang[19], a programming language developed by the telecommunications industry for environments with extremely high concurrency. It supports extremely high amounts of concurrent activity, and makes it much easier to avoid concurrency issues. Elixir runs on a virtual machine; programs are often shipped with the virtual machine included, resulting in a single binary that does not depend on any Elixir packages in the host system.

Advantages:

- Supports extreme amounts of concurrency, which is especially useful when proxying requests
- Shipped as a single binary, making installation simpler for the end user
- Reliance on processes as data storage makes it easier to keep track of the state of various sessions of authentication mechanism, and to avoid concurrency issues in these flows

Disadvantages:

- Elixir is a relatively obscure language, making it harder for other developers to contribute to or extend the service
- The virtual machine has dependencies that are not universal across different Linux distributions, which in practice means that the program has to be compiled for each operating system. This also means that both Erlang and Elixir need to be supported on these operating systems, although they could be compiled for said operating system if they are not available in the package manager.

Verdict:
While this service does not require sufficient concurrency to properly benefit from the features of Elixir, using Elixir has no major disadvantages.

5.3.7 *Decision*

A decision can be made by process of elimination. Python, Java, and JavaScript are not suitable for this project, as they can all lead to difficulties in installing the software for the end user, and because they exceed some resource requirement. JavaScript is further unsuitable because of potential security vulnerabilities. PHP is unsuitable because the language was designed around pure web projects, which means that providing LDAP and PAM Services violates the basic premise of the language and will require workarounds to implement, greatly increasing complexity. While C++ has been seriously considered, the author admits to not having enough confidence with the language to avoid crashes or security vulnerabilities associated with memory management.

This leaves Elixir as the language of choice. This has the further advantages of being shipped as a single binary, and the high concurrency allowing the system to scale well, which is especially important when providing proxying services. Furthermore, it makes it easier to ensure that the system remains online, which is especially important considering that without a functioning authentication service, all other services could become inaccessible. This disadvantage are that the binary ships with the Erlang VM, which uses a relatively large amount of disk space, and that the system might introduce more latency than other languages. Furthermore, this does mean that the service will have to be compiled again for each supported operating system, although this should be possible for almost all Unix-like operating systems.

This document does not posit that Elixir is the best possible language for this task; languages such as Rust or

Haskell are faster than Elixir, while avoiding the memory management issues that C++ has. However, the author is not sufficiently familiar with these languages to write such a service in them. This document does posit that Elixir is the most suitable language that the author is familiar with.

5.4 Functionality View

This view will elaborate on the requirements, by listing the functionality that will actually be provided. This view consists of an itemised list of features. This view is intended for potential end users, system administrators, system integrators, and anyone else interested in the project, so that they may obtain a concrete list of features that they can expect from the system; additionally, developers may use this as a checklist of tasks. Some functionality may only be provided given available time, corresponding to the optional requirements; this will be indicated by prefacing it with some form of the word "optional". Functionality will be divided into several categories, depending on which stakeholder it is intended for.

5.4.1 End Users

- Username/password authentication portal
- Addition of TOTP to this authentication portal
- The storage of limited personal data, i.e. username, name, email address
- Optionally, the storage of other profile data
- A form where the user can change their password
- A form where the user can enable and disable TOTP, and configure their TOTP device
- No record is kept of old user data
- OAuth authentication services
- LDAP authentication services
- Forward authentication services
- Proxy authentication services
- Optionally, SAML authentication services
- Optionally, PAM authentication services

5.4.2 System Administrators

Features in this list are only accessible to administrative users

- A place where client applications can be viewed, added, and removed
- Per client application, a place where the credentials of this client application can be viewed, and any configuration can be done
- Optionally, per client application, a place where users can individually be granted or denied access to the client application
- A place where users can be viewed, added, removed
- Per user, a place where the user can be viewed and their non-credential information can be edited. This includes whether or not the user is an administrator, with the exception that users cannot remove their own administrator status to prevent leaving the system in an inaccessible state
- Per user, a place where the administrator can reset the users credentials to some randomly generated value
- A method for the root user on the host system to reset the password of a user to a randomly secured value
- A configuration file where at least the following can be configured:
  - Whether or not the system integration socket should be enabled
  - The file location of the system integration socket
  - The file permissions to set on the system integration socket
  - The port on which the LDAP service should listen
  - The Distinguished Namespace in which LDAP users should appear to reside
  - The apparent name of the object class of a user
  - The name of the uuid field that LDAP entities appear to have
  - Which type of reverse proxy to run
  - The port on which the reverse proxy should listen
  - Whether or not to require that users who have TOTP enabled should append their TOTP code to their password for LDAP services
- Optionally, a settings page where the above can be configured through the web interface

5.4.3 System Integrators

See section 5.6

5.5 Component View

This view is intended for developers in order to obtain a technical overview of how the system works. It consists mostly of long form textual descriptions, but it is also visualised in a component diagram using UML lollipop notation. In this notation, the U-shaped socket indicates that a component provides a service that can be used by other components, and a circle inside of this socket indicates that a component relies on this provided service. Note that these two elements can also exist independently, in order to refer to a socket open to external programs and a dependency on an external program respectively.

DAS consists of several components. This section will briefly identify these components, and then show an overview of how they interact. Each component also has their own section, in which the component can be seen in detail. A schematic overview of this view can be found in figure 7.

The central component is the user database, which is responsible for storing the user data and providing it to other components. Related is the session manager, which is responsible for creating and maintaining HTTP session tokens, with are used to authenticate a browser to a user, which is then used by OAuth, SAML, and reverse proxy authentication. Note that this means the HTTP login portal is actually the responsibility of the session manager. Besides authenticating users, the service must also be able to authenticate client applications; storing the credentials of the client applications is handled by the client database. Both the client and user databases rely on the persistent storage component to ensure that their information can persist across

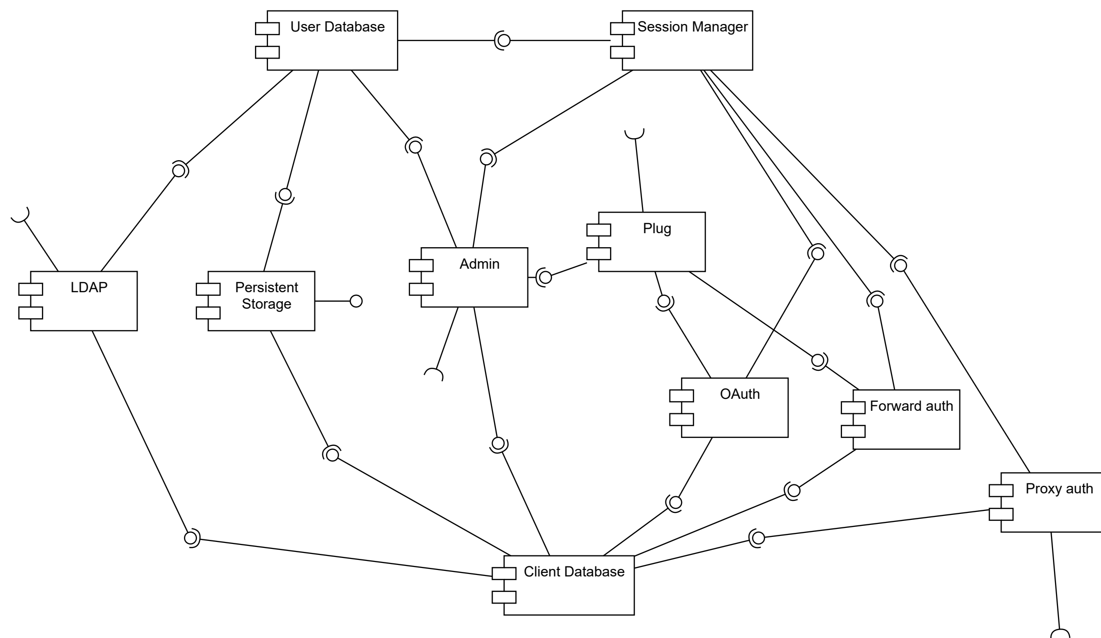

Fig. 7. A schematic overview of the components in the service, visualised using UML lollipop notation. The open sockets represent external connections.

restarts. Of course, we have not yet talked about the actual authentication protocols. OAuth will be handled by the OAuth component, which can be extended to also provide SAML services. LDAP will be handled by the LDAP component, which can be extended to provide a socket for PAM services as well. Forward authentication will be handled by the forward auth component. Proxy authentication will be handled by the proxy auth component, which will open an extra socket. Services for the administrative interface will be handled by the admin component. Web requests for OAuth, SAML, and the administrative interface will first come in at the Plug component, which is a generic HTTP service that will handle any HTTP specifics before forwarding the request to the appropriate component; because the socket for proxy authentication must run on a different port, web requests to this protocol will be handled by the reverse proxy authentication component itself.

Finally, the admin component is responsible for allowing the user to manage the service, which facilitates functionality such as adding and removing users and client applications. To allow home server automation systems to automatically configure client applications, this component can also open an external socket on which it can receive commands.

5.5.1 User Database

The User Database is not just responsible for storing user data, it is also responsible for ensuring the persistence of the data, as well as verifying any user credentials.

To minimise the amount of code that has access to user credentials, it should not be possible to export user credentials from this component. Instead, this component can only be used to verify credentials. In order to maintain logical consistency, this component should also be exclusively able to verify TOTP codes.

5.5.2 Session Manager

The session manager is responsible for two things. Firstly, it is responsible for maintaining session tokens, which are temporary values that associate a browser to a particular user. Secondly, it is responsible for creating these tokens, by providing the web pages interaction with which the user authenticates in the first place. With forward authentication, the validity of the token needs to be checked on every request; as such, performance is important for this component.

Essentially, this component provides three categories of external interface. Firstly, it provides services to validate and dereference existing tokens. This functionality can be accomplished with an Elixir Registry[52], and possibly a few calls to the User Database component. Secondly, it provides services to authenticate web browsers, making the session manager responsible for the backend components of the login portal. Finally, since the login portal is in this component. this is also the most logical component to be responsible for the rest of the user interface endpoints, i.e. the endpoints where users can view and change their own data.

5.5.3 Client Database

The client database is responsible for storing and providing information on all the client applications in the software. This includes the type of the application, any configuration information for how the client should be served, as well as the client application's credentials.

5.5.4 Persistent Storage

The persistent storage component is responsible for ensuring that information from the user and client databases is not lost if the service is shut down, crashes, or otherwise unable to maintain its own state. The data can be persisted in various ways; the two most notable ones are to write it

to disk, or to a database. Both carry the risk that the data may be read by unauthorised third parties, and in both cases this risk can only be mitigated by correctly setting up permissions outside of the control of the service itself. Of course, secure password hashing must be used to reduce the severity of such a breach.

Writing to the database is more performant, especially in high-concurrency scenarios. However, it is more difficult for the system administrator to set up, especially since it requires an already running database. As such, this component should support both methods of data storage, with the ability to configure which method to use. As this component is already required to interact with databases and as such required to use SQL, it seems logical to use SQLite for storing data to a file.

5.5.5 LDAP

This component is responsible for providing authentication services for protocols that request credentials themselves and forward them to the authentication service. It will initially focus on support for LDAP, with a possible expansion into providing bindings for PAM.

In order to provide LDAP services, this component must open a TCP socket on the appropriate port. When a client application connects to this socket, the component must perform the following services:

1) Confirm that the credentials of the client application are correct by communicating with the client database
2) Request information about the user from the user database
3) Send a response to the client application
4) If the client application responds, confirm that the provided user credentials are correct; if applicable, by splitting of the TOTP code and verifying it separately with the user database.
5) Send a response to the client application.

5.5.6 OAuth

This component is responsible for providing authentication services for protocols that use the web browser as the primary method of communication, i.e. OAuth and SAML. In order to take optimal advantage of the similarities between the two protocols, this component itself is split into three parts: one parts handles the syntax for OAuth, one part handles the syntax for SAML, and one part handles the common logic. Because the exact flow can vary for both protocols, the common logic simply provides some functions for what to do given the available information, and the individual protocol components are responsible for calling the right function at the right time. Note that actually authenticating the user is the responsibility of the session manager.

5.5.7 Forward Authentication

This component is responsible for providing forward authentication services. It needs to provide an endpoint that a reverse-proxy can connect to to check if a user is authenticated, and a method of authenticating unauthenticated users. As this endpoint will be accessed on every request to the client application, speed is important for this component.

5.5.8 Proxy Authentication

This component is responsible for providing proxy authentication services. It needs to open a new socket, through which it can proxy requests. It also needs to be able to make HTTP requests to the client application.

Since every request to the client application will be proxied through this component, speed is important. It's also important the component can correctly proxy the HTTP protocol, and does not have to wait for the client application to complete its request before returning a response.

5.5.9 Admin

This component is responsible for allowing the system administrator to administer the service. This effectively means making changes to the client and user database. This component can be accessed via two methods: a web interface, which is intended to be accessed by the system administrator, and a socket for use by system integrators.

The web interface of course also requires authentication, as only a subset of users should be able to access it. In order to facilitate this, this components accesses the session manager.

5.5.10 Plug

Plug refers to the library used to allow for web requests. It is responsible for ensuring that all web requests, whether they be intended for the OAuth, admin, or session manager components, reach their appropriate destination. It is also responsible for translating between HTTP requests and Elixir's internal data structure. This ensures that the rest of the system can communicate using native datastructures, instead of passing around serialised data with HTTP trappings.

5.6 System Integration View

This view provides API documentation of the provided socket for system integration. It is intended for system integrators, who may find this information helpful while integrating the service into their system. As the same endpoints are also availabe to administrative users in the web interface (although prefaced with `/admin`, this information is useful for (front-end) developers as well.

All interactions go over the system administration socket. As stated in section 5.4, the location and file permissions of this socket are configured via configuration file. This socket in fact provides an HTTP Server service a REST API, and it is recommended to interact with the socket using an HTTP client. It is strongly advised against making this socket accessible outside of the local machine for security reasons.

This section will now list API endpoints of the administration socket, what methods they accept, which actions these methods perform, which parameters they require, and what response can be expected.

For all API requests, the following response codes may be returned:

- 200: If the request was successful
- 201: If the request was successfully received, but has yet to enact change

- 400: If the request body was malformed, or is missing a parameter
- 404: If the requested endpoint is unknown, or does not support the requested method
- 500: In case of an internal error with the service; this may be a bug, but may also be due to the unavailability of e.g. a database server.

Unless explicitly specified otherwise, these are the only response codes that can be expected from any endpoint.

The following general basic datatypes are defined:

- `<type string>`: one of 'oauth', 'ldap', 'forward', 'proxy', 'saml', 'pam'
- `uuid`: long, random string used to identify a client application to the service. Serves as the client id in OAuth and SAML, and the Bind ID in LDAP. Format may vary according to the type of the client application

/client

Information about client applications. A client application is represent with a JSON object of the following format:

```
{
  'id': <uuid>,
  'name': <string>,
  'type': <type string>,
  'url': <string|null>,
  'destination': <string|null>
}
```

GET

Get a list of current client applications. The response body is a list of client application objects

POST

Add a new client application.

Request body has the format of the client application object. The `name` and type must be provided, and the `type` must be valid. The `destination` is only used for proxy authentication, and for proxy authenticated application must refer to the location where the client application can be reached. The `name` value is optional and only used for display purposes. The `id` value will be ignored if provided.

/client/<id>

Modify or retrieve information on a specific client.

GET

Returns a client application object describing the request client application

PUT

Make changes to a client application.

The request body must be a client application object. The `id` will be ignored if providded.

DELETE

Deletes the requested client application.

Any sessions with the client application will continue to function until they are terminated If any browser is in the process of authenticating while this action is called, the resulting behaviour is undefined.

/client/<id>/credentials

Used for interacting with the client credentials of one client. Uses the following format:

```
{
  'type': <type string>
  'id': <uuid>
  'secret': <string>
}
```

GET

Returns the credentials of the client id, in the format above

/client/<id>/callbacks

Used for interacting with callback uris. Callback uri's are a required security feature for OAuth applications, where every callback URI used by the client application must be registered under this endpoint. They are not used for any other protocol.

GET

Get a list of callback URIs for the specified client application. Returns a JSON list of strings.

POST

Add a callback uri. The body must be the callback URI.

DELETE

Delete a callback URI. The body must be the callback URI to delete. Returns a 404 the provided callback URI was never registered for the client in the first place, or 200 otherwise.

/user

This endpoint defines the following format for a user object:

```
{
  'id': <unique integer>,
  'username': <unique string>,
  'email': <unique string>,
  'name': <string>,
  'administrator': <boolean>,
  'totp_enabled': <boolean>,
  'totp_ldap': <boolean>
}
```

Additional keys may be included

GET

Retrieves a list of all users in the system.

Response body is a list of user objects

POST

Creates a new user.

Request body must be a user object. At least `username` and `email` must be provided. If they are not provided, `name` will default to `''`, and `administrator` will default to `false`. If `id`, `totp_enabled`, or `totp_ldap` are provided, they will be ignored.

If a username or email are given that are already used, this endpoint may return a 408 status code. Future functionality may disable the ability for administrator users to be created via the socket; in this case, a 401 may be returned if `administrator` is set to `true`.

/user/<id >

This endpoint interacts with a particular user. It uses the same user object format as describe above.

GET

Returns the user object for a particular user.

PUT

Updates the user entity.

The request body must be a user object. `email`, `name`, and `administrator` can be altered; any other keys will be ignored.

If a username or email are given that are already used, this endpoint may return a 408 status code. DAS will ensure there is always one active administrator. This action will return a 408 status code if trying to remove administrator status from the only administrator user. Future functionality may disable the ability for administrator users to be enabled or disabled via this socket; in this case, a 401 may be returned if trying to alter `administrator`.

DELETE

Deletes the user.

No request body is required.

DAS will ensure there is always one active administrator. This action will return a 408 status code if trying to delete the only active administrator.

/user/<id >/change_password

PUT

This endpoint can be used to change the credentials of the user. Calling this endpoint will also disable TOTP for the user.

There is no request body; the caller can't choose the password. Instead, the service will generate a new password, and this will be returned in the following format:

```
{
  'password': <string>
}
```

The user can then change their password through the regular interface.

5.6.1 /client_ldap_area

Returns the DN of the LDAP folder in which the clients and users appear to be. Used by the front end to correctly display an LDAP DN for the client ID.

GET

Returns the LDAP Area as a string; this endpoint does not return a JSON object.

5.7 Testing View

This view describes how DAS will be tested, verified, and evaluated. It is intended to be used by anyone who would like assurance as to the quality of the service, as well as developers. It is split into three sections: a section which describes the environment that will be used to developed the service in, a section which will describe how the quality of the code is verified, and a section which describes how the quality of the completed system will be evaluated.

5.7.1 Unit Testing

This section describes how the direct quality of the code will be verified, i.e. how it will be tested that the actual behaviour of program components is the same as their intended behaviour. In accordance with Elixir best practices[12], this will be done by writing Unit tests for all functions that are exported by any included components.

To prevent the unit tests from having side effects, such as interaction with a database or a TCP port, certain components will be mocked. In accordance with best practices, this will be done by ensuring that all calls to these components will read the name of the component from the program configuration, which allows them to call a mock component instead of the actual component during testing. This mock component will return data that the actual component could also return, but without performing the side effects.

5.7.2 Development Environment

Application	Protocols	Reason
Nextcloud[39]	OAuth, LDAP	Most popular application
Samba[1]	LDAP	Non-web based program
Sonarr[56]	Reverse-proxy	Most popular reverse-proxy
Paperless-ng[61]	Reverse-proxy	Small reverse-proxy project
Ampache[2]	LDAP	Small LDAP project
Kanboard[26]	OAuth	Small OAuth project

TABLE 2
Applications selected for the development environment

This section describes the environment against which the service will be developed. During development, it is useful to have an environment similar to one that would be used in practice, including a similar set of client applications. This makes it much simpler to test code while it is being developed, and helps account for client applications that do not properly adhere to the standard. This reasoning does assume that the development environment provides an accurate and complete reflection of the practical environments.

It is known that a majority of domestic self-hosted systems use Debian Linux[7], and as such this is the preferred operating system for the development environment as well. Similarly, Nginx is the most popular reverse-proxy, and MySQL is the most popular database.

To provide a sufficiently broad selection of services, there should be at least two types of client application for each of the protocols: one that has large amounts of funding and can be expected to adhere to standards properly, and one smaller project that may have taken some shortcuts in their implementation. To ensure that these applications accurately represent the applications that one may be expected to find in real life, the most popular application that fits the criteria should be taken.

The applications that have been selected, as well as the authentication protocols for which they can be used and a small explanation for why they were selected, can be seen in table 2.

5.7.3 Evaluation

In order to evaluate the system, an early version will be deployed on a system that the author administers, which has a wide variety of applications and about 7 users. Furthermore,

the Yunohost development team will be consulted for any decisions that specifically impact system integration.

6 Implementation Details

This section will mention any significant implementation details that do not trivially follow from section 5. It will do so by going over each component, as describe in section 5.5.

Due to time constraints, SAML and PAM were not implemented.

6.1 Persistent Storage

Database access is provided by the Ecto library[17], which allows for the composition of data schemas and queries in Elixir code. It supports many different ways of actual storage via a plugin system.

The implementation for persistent storage simply holds three types of data repository; one for MySQL, one for PostgreSQL, and one for SQLite. When queried, it returns the one that has been selected by the configuration file.

6.2 User Database

The user database defines a database schema for a user object, and a module with basic CRUD operations. This module also includes the notable function `verify`, which takes in a username/password combination and returns a user struct if and only if the password matches the username. The user object has an id property, which is a unique unchanging integer; an email property, which is a unique, but not unchanging string, that is not verified to be an email in any way; a name property, which is a generic string; a password property, which is a hash of the users password; optionally, a totp secret; a boolean to indicate whether or not the user is an administrator, and a boolean to indicate whether or not TOTP over LDAP should be enabled for this user.

6.3 Client Database

The client database defines a database schema for a client object, and a module with basic CRUD operations. A notable feature of the schema is that the primary key is not an incrementing integer, but a 16-byte random binary value; this value can also be used as a client id for OAuth.

Which protocol the client application supports is indicated by one field in the schema. This field is only used for display purposes; any registered client application could, given the correct information, use any protocol. The client database provides a `verify` function similar to the that of the user database, although client secrets are not actually stored as hashes.

There are four generic fields that are always filled in: the client id, the client secret, the client name, and the client url. The client id and secret are used by OAuth and LDAP; the client id is also extremely important internally as the primary identifier for a client application. The name is used solely for display purposes. The URL is used for display purposes by all types of client application, but also used as a means of identification for client applications that use forward or proxy authentication. Additionally, there is the destination field, which is used solely for proxy authentication, as the location to proxy requests to.

There is additionally the callback URIs table, which is used solely for OAuth client applications. This table stores the registered callback URIs; if an OAuth client application attempts to redirect a user to any other callback URI, this operation will not succeed for security reasons.

6.4 LDAP

LDAP uses a binary data exchange format, which is defined in the ASN.1 language. Parsing this format was done by using the `asn1ct` module from OTP, which can compile ASN.1 definitions into an erlang module that encodes and decodes the binary data. The ASN.1 definitions for the LDAP wire protocol, as well as the header file for the parses, were taken from the source code of the `eldap` erlang module. Because the resulting code was erlang, it was put into a separate erlang code project in a directory, which was imported as a source-code dependency to the main project.

The implementation of the LDAP component is quite different from that of a standard LDAP server, mostly because the service is not actually an LDAP server. While user data can be accessed via the LDAP wire protocol, it can not be modified via this protocol in any way, and the data does not actually reside in a directory structure.

This means there are impactful decisions about how to translate the single database table that stores user data to the directory structure expected by the client application. The most trivial way is to pretend that there is only one directory that holds all the data, and that all other directories are empty. DAS, however, uses a different method: it pretends that the data is in all directories, by ignoring all but the least significant parts of location (or, in LDAP terms, it ignores all but the first part of the DN). This applies to everything in the client request, as well as the name the client uses to authenticate itself (in LDAP terms, the Bind DN). The advantage of this approach is greater support for misconfigured clients and clients which do not correctly implement the LDAP specification; any client will always be looking for users, and this approach ensures they will always find the client for which they provided a username. The Bind operation applies to both clients and users; which entity type is being checked against does not depend on the location of the entity, but on whether an `id` is provided (for clients) or a `username` (for users). The directory that users appear to be in for any results do not depend on the directory used by the input, but on a value defined in the configuration file.

Not all request types in the LDAP protocol[65] are supported. The supported request types are Bind, Unbind, Search, and Compare; Search and Compare are only available if the session is authenticated as a client. The Add, Delete, Modify, ModifyDN, and Extended request will fail with a result code indicating insufficient access rights, but with an error message stating the request is not supported. The Abandon operation is simply ignored; this request is normally used to cancel an ongoing search if it takes too long to process the results, but it is not expected this will be a problem for this service, as it only queries one database table.

The code that handles the Search request must transform the search query into a SQL query. This is not possible using the normal Ecto syntax, which needs to be processed at compile time; this processing does not work if the full query needs to be built based on external input at runtime. Fortunately, the internal datastructure that Ecto uses to store a database query is remarkably similar to the structure of the Filter datastructure used by LDAP, allowing the code to transform the LDAP request into a database query by creating an Ecto internal datastructure and injecting it into a query struct.

Any filter in a search query that refers to an attribute of the entity that does not exist is ignored; this does not mean it is implied to be always true or always false, but simply that these filters are not included in the SQL query that is generated from the request. This is done because many implementations filter not just on the username of the user, but also on the `objectClass` attribute, which would normally distinguish user entities from other types of entities in the system. Since this service will only return user entities under any circumstance, this attribute can be simply ignored. Instead of ignoring only the `objectClass` attribute, a more general approach has been taken, and all filters for unknown attributes are being ignored. This does not apply to the `present` filter type, which returns true if and only if the attribute is part of the users schema.

6.5 OAuth

The OAuth implementation is entirely standards-compliant, although some features are still unimplemented due to time constraints. For example, PKCE has not been implemented. Furthermore, only the "code" and "id_token" flows are currently supported.

The implementation of authorization codes bears mention. The authentication codes are generated by the authentication provider, and sent to the client application via the user's browser. The client application then exchanges them for an access and/or ID token. The codes should refer to some data on the transaction, and only be usable once and for a short amount of time after they have been generated, and Elixir lends itself uniquely to this problem. The implementation uses the Elixir Registry to accomplish this behaviour. The Registry is a cross-process lookup table that associates a value with a process. The association only holds so long as the associated process is alive. The service defines a process that holds all the information associated with an authorization code (e.g. the authenticated user, or the requesting client). This process will listen for 5 minutes to see if it gets any messages that ask for the data the process holds. If it receives such a message, it will return the data and exit. This ensures that even under the strangest race condition, an authorization code can only be redeemed once. Access tokens are stored in a similar fashion.

The RSA keypair used for signing the ID token is stored in a special table in the database. When the application starts, it checks if these values are already present; if either the public of private key is missing, both are freshly generated and stored. The keys are stored in the database in PEM format. To avoid having to parse this for the required values on every request, they are parsed into the required format at application startup and stored in memory.

All callback URIs have to be registered. This may cause issues for users, as many client applications don't explicitly declare their callback URI. To make it slightly easier for system administrators to register the callback URI, the error message that is displayed to the user when the callback URI is invalid contains instructions on how to register the callback URI. Only mandating callback URI registration for apps using the implicit flow should be taken into consideration.

6.6 Forward Authentication

The DAS endpoint that handles the authentication check is extremely simple: it simply checks if the user has a session, and if so, returns an OK response with some headers that contain values from this session. Specifically, the username, email, and name are put into headers. As this endpoint is called on every request to the client application, it is important this process is as fast as possible; as such, the whole user object from the database is put into the session. This does have the disadvantage that upon updating user data, old data will persist in old sessions.

The larger problem is how to establish a user session. Any request that enters the proxy authentication endpoint at the authentication provider is a forwarded request that was originally made to the client application, and is as such likely to be under a different domain than the main DAS domain, meaning that established session cookies from the regular DAS login portal do not persist. As such, clients will have to be authenticated in a separate way from the normal login portal.

The most trivial solution to this problem would be to simply show the login portal for any denied requests, and have the user establish a session in this manner. However, this has two major drawbacks. Firstly. the response served to the user for denied requests is (at least when Nginx is used) determined by the reverse proxy, and configuring it to correctly serve the login portal, as well as correctly handle the subsequent requests required to interact with the login portal, would have to be done under the domain of the client application; this would lead to complex reverse-proxy configurations. Secondly, in this approach the user would have to enter their credentials for every new reverse-proxy authenticated service, even if they already established a session with the main DAS interface.

To avoid both of these drawbacks, a different approach was taken. If the user makes an unauthenticated request to the client application, they will be redirected to the a special DAS endpoint under the DAS domain, which will be referred to as the session creation endpoint, This endpoint will check if a session has already been established, and refer to the login portal if not; after the user has logged in, the login portal will redirect back to the session creation endpoint. The session creation endpoint will then reuse the authorization code feature from OAuth, and create an authentication code that contains the necessary user info. It will then redirect the user back to the original request they made to the client application, but with the authorization code as a query parameter (in addition to any query parameters that were in the original request). The reverse-proxy endpoint will then redeem the authorization code and

establish a session under the domain of the client application. This process is essentially a basic implementation of the OAuth implicit flow where the reverse-proxy endpoint acts as an OAuth client application. Normally, the implicit flow in OAuth it is not possible for the authentication provider to verify the identity of the client application; however, this is not an issue in this case, as the client application and authentication provider are both DAS running under two different domains. To make it more difficult for an attacker to intercept authentication codes, it is verified that the authorization code was generated in a request that redirects to the url that is requested when it is redeemed, and the client that is attempting to use the code has the same IP address as the one that the code was generated for. The session creation endpoint also verifies that the URL that the user was redirected from actually corresponds to a registered reverse-proxy application. IP verification is skipped if the code was generated for an IP address with a different version as the one trying to redeem it, because of an issue encountered during testing where verification would incorrectly fail; the server was reachable over both IPv4 and IPv6, and the browser used IPv4 to contact the client application, but IPv6 to contact the DAS portal, failing the verification because of a different IP address.

As previously mentioned, the DAS forward authentication endpoint does very little; instead, much of the logic is handled by the reverse-proxy. This means the reverse-proxy needs to be configured correctly for this type of authentication to work. Concretely, the reverse-proxy needs to be configured to do the following:

- Before any request to the client application, first forward the request to DAS. To improve performance, the request body should be ommitted from the forwarded request.
- If DAS returns a 401 response, return a 302 response to the user, redirecting them to the session creation endpoint of the authentication provider
- If DAS returns a 200 response, make the request to the client application; make sure to forward the `Remote-User`, `Remote-Email`, and `Remote-Name` from this response to the client application.
- If DAS returns a 200 response, make sure any `Set-Cookie` headers from this response are forwarded to the user.

In order to aid system administrators who use Nginx, a folder with various configuration snippets that configure Nginx to the above is provided along with the service, as well as example configurations that use these snippets.

6.7 Proxy Authentication

Proxy authentication establishes a session with the user in the same way as forward authentication, and even reuses the session creation endpoint. Just like with forward authentication, the whole user entity is stored in the session, in order to improve performance.

The proxy authentication code needs to know where to forward the request. The location must consist of a scheme (either HTTP or HTTPS) and a host, with optionally a port. While the location is stored in the database, in the `destination` field of the client application, querying the database for every request would lead to unacceptable latency. Instead, a Registry is used to couple the `Host` header of the HTTP request to the location of the client application. The Registry is updated every time the `destination` of the client application is changed. The values stored in this Registry are already parsed into separate scheme, host, and port values.

Requests are made via the Mint library[18]. If the user is authenticated, the incoming request is forwarded to the client application as is, although the `Remote-User`, `Remote-Email`, and `Remote-Name` headers are overwritten with the values of the corresponding user, and the `Host` header is overwritten with the host part of the request destination. The response from the client application is forwarded back to the client application with no alterations, save for one: the `Transfer-Encoding` will always be set to `Chunked`. This is because the authentication provider may receive the response from the client application in several parts, and passes these on to the user without buffering. This functionality requires the use of the Plug chunked sending feature, which is applied to all requests.

6.8 TOTP

The implementation of TOTP relies on external libraries for both the logic and the generation of QR codes. A notable implementation decision is that the QR code for setting up a TOTP authenticator app can be viewed at any time after TOTP has been enabled. Disabling TOTP will delete the secret from the application, and a new secret will be generated if TOTP is re-enabled.

Whether the user has TOTP enabled is included when viewing the user object, but it can't be edited via the basic CRUD endpoints, by either the user or the system administrator. It can only be enabled and disabled via a special endpoint. If the administrator resets a user's password, this will automatically disable TOTP; this is also the only way to disable TOTP when a user has lost access to their authenticator app. This is a conscious design decision aimed at preventing social engineering attacks.

When using TOTP over LDAP, the client application, and by extension the login interface, are not aware that TOTP is being used, and will not communicate this to the user. Instead, the user is expected to remember this by themselves, and append the TOTP code to their password (with no separator). To add to the inconvenience for the user, the client application does not display the reason for authentication failure, and as such the user can't be reminded to enter the TOTP code. Furthermore, if the authentication does not succeed, the user is unable to determine whether this is due to a wrong password or a wrong TOTP code. As such, TOTP over LDAP can be enabled by each user separately from TOTP, and is disabled by default.

6.9 Static files

To save effort and maintain a fully RESTfull API, the decision was made not only serve HTML as static files, and force the front-end to use REST API calls to retrieve any relevant data. This means that the HTML is completely devoid of data, but includes JavaScript which retrieves the data from

the correct endpoint. This logic extends to the login page; the login request is done via JavaScript, and it's the JavaScript which then redirects to the appropriate page.

All the static files are stored in the same directory, and all have file extensions; there are no API requests with a file extension. An optimally efficient deployment would configure the reverse-proxy to serve these files directly for all requests which end on a file extension. However, to ensure maximum compatibility and allow for much simpler reverse-proxy configuration in deployment, these files are also served by the service itself for the appropriate requests.

6.10 Front-end

The front-end webpages implemented entirely RESTfully, i.e. the original HTML contains no dynamic information, but instead this information in filled in by JavaScript based on the results of some API calls. The pages have been styled using the Pure.css[25] framework; this framework has been chosen because of its exceptionally light weight, and the relative ease with which it allows for making both desktop and mobile websites.

The most important page of the front-end is the login page, which is used not only to log in to the service itself, but also used by the OAuth, Forward Authentication, and Proxy Authentication flows. The page itself is extremely simple, containing only one form that tries to log the user in and redirects appropriately upon success. If the user has two-factor authentication enabled, the page will ask for the users two-factor authentication code without reloading the page.

Apart from the login page, there are three pages: the page where users can change their own data, the page where administrators can administer users, and the page where administrators can administer client applications. The latter two of these page are only accessible to users who are administrators.

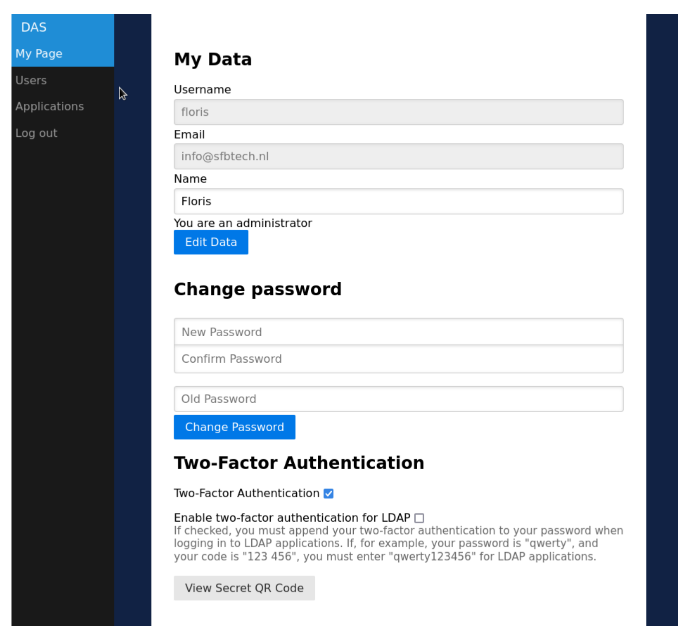

Fig. 8. The home page of the DAS interfacet

The "My Page" page allows users to view their own data, and change their name, password, and enable or disable two-factor authentication. Figure 8 shows what this page looks like.

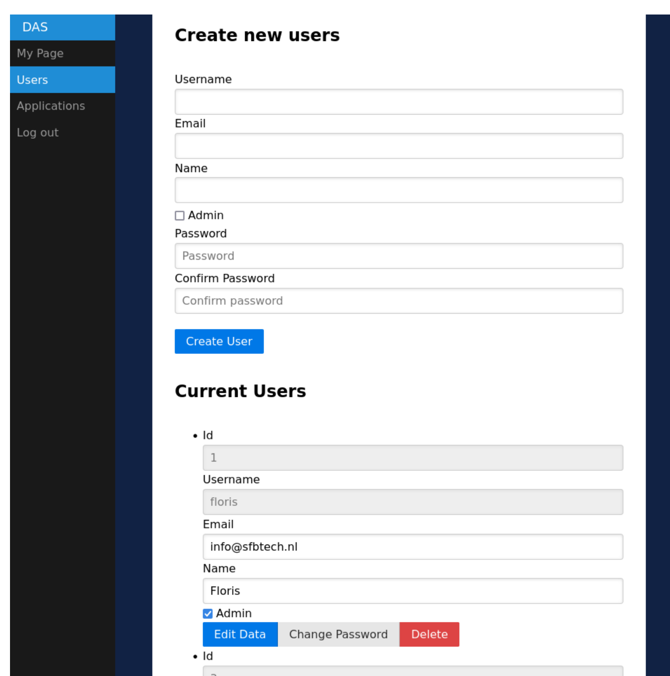

Fig. 9. The page where administrators can administer users

The "Users" page allows administrators to create new users and update or delete existing ones. It also allows the administrator to create new administrators, or to change users passwords to a temporary random value. Figure 9 shows what the "Users" page looks like.

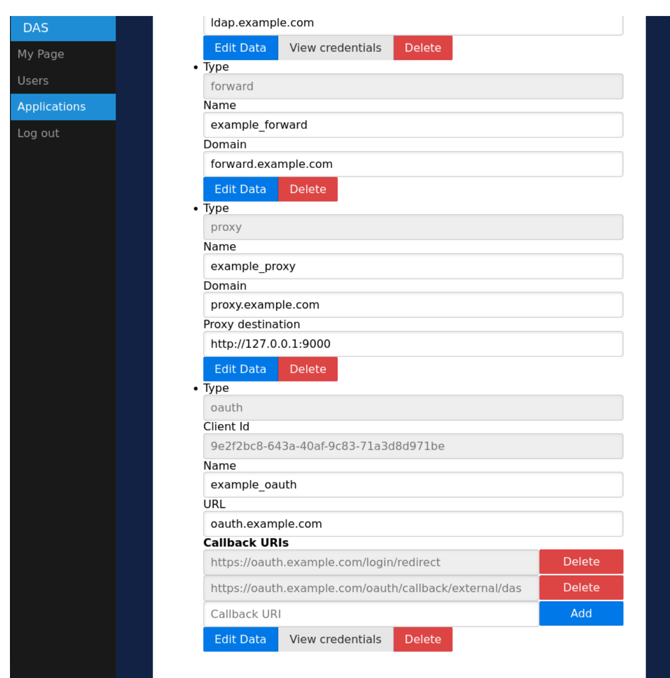

Fig. 10. The page where administrators can administer client applications

The "Client" page allows administrators to administer client applications and update or delete existing ones.

The interface that is presented to the administrator differs somewhat per type of client application. For LDAP client applications, the BIND DN is presented instead of the ID. For Proxy client applications, the proxy destination is to be configured in addition to the other values. For OAuth client applications, callback URIs are to be additionally configured.

Figure 10 shows what the "Clients" page looks like.

6.11 Practical Protocol Comparison

This subsection will compare the four supported protocols as supported by DAS, discussing aspects that concern the practical deployment and have not yet been discussed in section 4.7.

The first of these aspects the complexity of configuring a client application with the correct credentials. An OAuth application, in the best case scenario, only needs to be informed of three values: the client id, the client secret, and the automatic discovery URL. However, not all applications use the automatic discovery URL, and is this case the application needs to additionally be configured with the authorization, token, and userinfo URLs, as well as the scope, bringing the total amount of required configuration settings to 6. In either case, DAS will have to be additionally configured with the callback URL. An LDAP client needs to be configured with the URL of the LDAP server, the bind DN and password, the default search DN, object class, and filter, which is also 6 values; however, it should be noted that DAS ignores the search DN and object class. Either way, configuring LDAP is less complex than configuring OAuth, as one only needs to generate the bind DN and password, and then only needs to configure the client application, in contrast to having to configure the callback URL with DAS as well. Forward and proxy authentication are considerably more complex to configure, as they require configuration of the client application and the reverse-proxy. In both cases, the client application needs to be configured with the name of the headers in which the user information will be transmitted; this usually amounts to two values, one for the username and one for the email. For forward authentication, configuration of the reverse-proxy is by far the most complicated configuration discussed yet, and will usually require combining the configuration file suggested by the developer of the client application for the reverse-proxy with the one suggested by the authentication service. For proxy authentication, this configuration can be simpler if the client application can be accessed entirely via an HTTP socket, and does not require further reverse-proxy configuration. If this is not possible, as is the case for e.g. PHP and UWSGI applications, a second reverse-proxy will need to be configured between the authentication service and the client application; this means the request will first travel to the reverse-proxy to distinguish it from other HTTP services on the same server, then through DAS for authentication, then through the reverse-proxy a second time to translate the request to FastCGI/UWSGI/etc, before finally arriving at the client application. This is obviously the most complex to configure, and may also incur a significant performance penalty.

Another important aspect is the performance, which we can define as the amount of time it takes the user to be authenticated to the client application, given that they are already authenticated with the authentication provider. There are several factors that introduce latency to this process: database queries, computation, data exchange between the client application and the authentication provider, and data exchange with the user. In a domestic self-hosting environment, we can assume that the client application and the authentication provider are running on the same machine, but that any communication with the user must go over a network call. As such, we will assume that the amount of requests the user has to make is far more significant than any other factor. We shall count the amount of network requests after the client application realises the user is not authenticated, and until the user is authenticated with the client application. LDAP requires one user request to the client application containing the user credentials, and is thereby the fastest. OAuth, forward authentication, and proxy authentication require two network requests: one towards DAS to generate an authentication code, and one towards the client application to retrieve this code. In case of OAuth using the id tokens from OpenID Connect, this second request contains the user information, and as such completes the flow. In the case of forward or proxy authentication, this request will be forwarded to DAS before being sent to the client application, which introduces a slight delay compares to the OpenID Connect flow. In the case of base OAuth, the client application must separately contact the authentication provider, which should introduce comparable delay. As such, LDAP has the highest performance, followed by OAuth using the OpenID Connect flow, followed by a shared third place between forward authentication, proxy authentication, and base OAuth.

Another important performance metric is the performance after the user has been authenticated. While this is outside the control of the authentication provider for LDAP and OAuth, it is extremely relevant for forward and proxy authentication, where DAS must authenticate every request. This necessarily takes time, and as such forward and proxy authentication will always slow down a client application when compared to OAuth or LDAP. It is unclear whether forward or proxy authentication adds more delay; while proxy authentication does require DAS to process the full request, it does not require a response to be returned to the reverse-proxy before being processed. What can be said, however, is that the worst-case scenario for proxy authentication, where the request travels through a second reverse-proxy before reaching the client application, is very likely to incur the greatest performance penalty.

From the above comparisons, it may look like LDAP is the most ideal protocol, as it is the easiest to configure and the most performant. However, it does suffer from a major drawback in convenience to the user, as it is the only protocol where the user has to enter their credentials for every client application, even if they are already authenticated with DAS. Furthermore, using TOTP in combination with LDAP leads to even more inconvenience, as described in section 6.8. Instead, OAuth should be considered the most ideal protocol, as it is unique in both allowing DAS to control the login portal, and not affecting the performance of the client application after the user has been authenticated.

7 Results Achieved

Section 1 asked 5 research questions, which have been answered by the rest of the document. This section will review these questions, list a summary of their answers, and direct to where they have been answered.

What are the differences between OAuth, LDAP, Reverse-proxy, SAML, and PAM?

OAuth and SAML function analogously, as do LDAP and PAM. Reverse-proxy should be split into two separate protocols, forward authentication and proxy authentication. as such, there are essentially four types of protocols. As in the rest of the document. SAML will be grouped under OAuth and PAM will be grouped under LDAP.

The main distinction between the four protocols is the flow of data between the authentication provider and the client application. This data includes the user credentials, which have to be communicated to the authentication provider, and the information on the authenticated user, which has to be communicated to the client application. In LDAP, the user credentials are first transmitted directly to the client application, which exchanges them for user information with the authentication service. In OAuth, the user first transmits their credentials to the authentication provider, then receives a one-time use code, which is transmitted to the client application via the user's browser, and then exchanged for user information with the authentication provider. In forward authentication, the credentials are transmitted to the authentication provider through the reverse-proxy, which then transmits the user information to the client application. In proxy authentication, the credentials are transmitted to the authentication provider, which then transmits the user information to the client application, along with the rest of the user request.

More information on this topic can be found in section 4

What are the challenges involved in making a service that can natively support multiple protocols, most notably OAuth, LDAP, and reverse proxy?

Implementing a service that can handle all four protocols is not much more difficult than implementing all four protocols separately. The main challenge is in creating a product line; that is, understanding the protocols and definining their commonalities and distinctions, such that code duplication can be avoided. This has been resolved by making a single datastructure for all protocols with unified CRUD endpoints, but different handlers for each protocols.

An additional challenge is posed by LDAP, which is designed to browse a directory structure; this poses a problem, given that none of the other protocols are designed to interact with directory structures. This has been resolved by ignoring the directory part of the LDAP queries, and always returning the any user entities that match the search.

More information on the exact nature of these challenges, see section 5.5; for information on how they have been resolved, see section 6

What are the challenges involved in ensuring such a service is suitable for domestic self-hosting?

The main challenges involved in making DAS suitable for domestic self-hosting lie in compatibility. This compatibility has several aspects. Firstly, domestic self-hosting has a wide variety of different system configurations, and any service should be able to deal with as many of these environments as possible; for example, the service can operate with MySQL, PostgreSQL, and SQLite databases. The ability to offer reverse-proxy authentication even under Apache, which does not normally support it, is another great example of this kind of compatibility. Secondly, DAS should also be compatible with several usecases, especially with a service that aims to interact with other services. This also implies that one should leave decisions up to the user as much as possible; for example, the only password restriction imposed by the service is a minimum length, and the value of this minimum length is adjustable via configuration option. The application can also be configured to listen on any port or Unix socket, and some features can be disabled via the configuration file entirely. Finally, one should also take hardware compatibility into account; this could mean offering releases for both x86 and ARM, but also means that one should take into account that the hardware may be outdated and not particularly powerful.

An additional note that should be taken into account is ease of installation; since the system administrator may be inexperienced, it should be as easy as possible to install the application. In this case, this means the service ships as a single binary, which opens a single HTTP socket to which a reverse-proxy can be connected. It also serves static files, in order to allow for the simplest possible reverse-proxy configuration. Finally, it is important that the application ships as a single binary without relying on too many external dependencies and environment configuration.

Some examples of the unique requirements of domestic self-hosting can be found in section 5.1, and section 6 contains some details on how this research addresses these requirements.

What are the differences between a self-hosted authentication service and its enterprise counterparts

DAS is missing several features of its enterprise counterparts. Most notably, DAS does not support any mechanism that would allow a specific user access to one client application without allowing access to all applications, such as realms or access control lists. DAS also does not offer any way for its client applications to distinguish groups of users. More information on features can be found in section 5.4.

Another important distinction between DAS and its enterprise counterparts is that DAS both provides authentication protocol services and holds user data in an integrated manner, and does not support any methods of separating this data. Enterprise services sometimes separate holding user data and providing authentication protocol services into two different applications, with LDAP being used as a protocol between the two services; this can be seen in e.g. Keycloak[27].

The protocols which are commonly used in domestic self-hosting are also notably different from those used in enterprise hosting. Enterprise authentication providers emphasise support for SAML[6], which is quite uncommon in domestic self-hosting. On the other hand, forward authentication support is quite rare for enterprise authentication providers, whereas there are many application common

in domestic self-hosting that exclusively support forward authentication. DAS is also, to the best of my knowledge, the only application that offers an alternative to forward authentication that works with the Apache reverse-proxy. There is also a unique technical aspect to DAS; to the best of my knowledge, DAS is the only application that provides authentication services for multiple protocols using a single datastructure for the client applications in all protocols. More information on the unique technical aspects and features of DAS can be found in section 6.

Finally, DAS can operate in much more barebones environments than its enterprise counterparts, being able to run on most Unix systems without requiring any language runtime, container system, or database program. It is also extremely lightweight compared to its enterprise counterparts, requiring little disk space and little memory.

What authentication schemes (e.g. traditional username/password, TOTP, asymmetric) can operate within the commonly used protocols?

Because of the limitations of LDAP (and PAM), only username/password is fully suitable. Any other solution that can authenticate a web browser can be used for OAuth, SAML, forward authentication, and proxy authentication. Theoretically, LDAP supports all solutions so long as they do not require more than a single transmission of two variable-length string fields; in practice, however, many such solutions would require modification of the client software, which would greatly diminish compatibility and are as such not suitable for domestic self-hosting. TOTP can used across all protocols as a second factor, but doing so when LDAP is used is inconvenient for the user, as the user must remember that TOP is enabled and will not be able to distinguish between a wrong password and a wrong TOTP code.

This question is primarily discussed in section 4.8. Some discussion on the practical implementation of TOTP can be found in section 6.8.

8 Conlusion

This research has produced DAS, an authentication service which supports OAuth, LDAP, and Reverse-proxy authentication, and is specifically tailored for domestic self-hosting. In doing so, it has establised some ground rules for writing software for domestic self-hosting.These rules can be summarised under three categories: compatibility with the variety of domestic self-hosting technologies, ease of installation and maintenance, and lack of features targeting large userbases. Furthermore, DAS is, to the best of my knowledge, the first service to approach the problem of multi-protocol authentication by defining a single datastructure to define client applications in any protocol. To the best of my knowledge, it is also the first service to allow for reverse-proxy authentication in combination with the Apache reverse-proxy, and also the first service to allow users to authenticate with TOTP-based two-factor authentication for LDAP-based services.

References

[1] *About Samba*. 2022. URL: https://www.samba.org/ (visited on 2022-01-17).

[2] Ampache. *Ampache - Music Streaming Server*. 2023. URL: https://ampache.org/ (visited on 2023-01-26).

[3] Lorenzo Angeli et al. "Conceptualising Resources-aware Higher Education DigitalInfrastructure through Self-hosting: a Multi-disciplinary View". English. In: *Eighth Workshop on Computing within Limits*. 2022.

[4] *Apache support - uWSGI 2.0*. 2016. URL: https://uwsgi-docs.readthedocs.io/en/latest/Apache.html (visited on 2023-01-27).

[5] The Vouch Proxy Authors. *Vouch Proxy*. 2022. URL: https://github.com/vouch/vouch-proxy (visited on 2022-11-27).

[6] BeryJu.org. *Making authentication simple*. 2022. URL: https://goauthentik.io/ (visited on 2022-10-07).

[7] Floris Breggeman. *Data from the Self-hosting survey 2021*. 2022. URL: https://github.com/florisbreggeman/selfhosting-survey-2021 (visited on 2022-10-27).

[8] Conor P. Cahill et al. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite*. Oasis Standards Working Draft sstc-saml-core-errata-2.0-wd-07. OASIS, Sept. 2015. URL: https://www.oasis-open.org/committees/download.php/56776/sstc-saml-core-errata-2.0-wd-07.pdf.

[9] Conor P. Cahill et al. *Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite*. Oasis Standards Working Draft sstc-saml-bindings-errata-2.0-wd-06. OASIS, Sept. 2015. URL: https://www.oasis-open.org/committees/download.php/56779/sstc-saml-bindings-errata-2.0-wd-06.pdf.

[10] Conor P. Cahill et al. *Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite*. Oasis Standards Working Draft sstc-saml-metadata-errata-2.0-wd-05. OASIS, Sept. 2015. URL: https://www.oasis-open.org/committees/download.php/56785/sstc-saml-metadata-errata-2.0-wd-05.pdf.

[11] Conor P. Cahill et al. *Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite*. Oasis Standards Working Draft sstc-saml-profiles-errata-2.0-wd-07. OASIS, Sept. 2015. URL: https://www.oasis-open.org/committees/download.php/56782/sstc-saml-profiles-errata-2.0-wd-07.pdf.

[12] Sean Callan. *Testing*. 2021. URL: https://elixirschool.com/en/lessons/testing/basics (visited on 2023-01-26).

[13] Scott Cantor. *SAML V2.0 X.500/LDAP Attribute Profile*. Oasis Standards Draft draft-sstc-saml-attribute-x500-01. OASIS, Oct. 2006. URL: https://www.oasis-open.org/committees/download.php/20650/draft-sstc-saml-attribute-x500-01.pdf.

[14] Oracle Corporation. *Oracle Java*. 2021. URL: https://www.oracle.com/java/ (visited on 2022-01-13).

[15] Debian. *LDAP PAM Authentication*. 2021. URL: https://wiki.debian.org/LDAP/PAM (visited on 2022-12-31).

[16] *Dovecot — The Secure IMAP server*. 2022. URL: https://www.dovecot.org/ (visited on 2022-01-14).

[17] Ecto. *Ecto v3.9.4*. 2023. URL: https://hexdocs.pm/ecto/Ecto.html (visited on 2023-02-25).

[18] elixir-mint. *Mint*. 2023. URL: https://github.com/elixir-mint/mint (visited on 2023-04-10).

[19] erlang. *Practical functional programming for a parallel world*. 2023. URL: https://www.erlang.org/ (visited on 2023-01-05).

[20] Fedetask. *Writing a Linux PAM module*. 2019. URL: https://web.archive.org/web/20190523222819/https://fedetask.com/write-linux-pam-module/ (visited on 2022-12-31).

[21] Apache Software Foundation. *Apache HTTP Server Project*. 2022. URL: https://httpd.apache.org/ (visited on 2022-01-13).

[22] OpenLDAP Foundation. *OpenLDAP*. 2022. URL: https://www.openldap.org/ (visited on 2022-09-26).

[23] *Gitea: Git with a cup of tea*. 2022. URL: https://gitea.com/ (visited on 2022-01-28).

[24] npm Inc. *About npm*. 2023. URL: https://www.npmjs.com/about (visited on 2023-01-05).

[25] Yahoo Inc. *Pure.css*. 2014. URL: https://purecss.io/ (visited on 2023-05-21).

[26] *Kanboard*. 2023. URL: https://kanboard.org/ (visited on 2023-01-26).

[27] KeyCloak. *Open Source Identity and Access Management*. 2022. URL: https://www.keycloak.org/ (visited on 2022-09-29).

[28] Keycloak. *Server Installation and Configuration Guide*. 2022. URL: https://www.keycloak.org/docs/latest/server_installation/#_database (visited on 2022-09-07).

[29] Martin Kováčik. *Using the NGINX Auth Request Module*. 2017. URL: https://redbyte.eu/en/blog/using-the-nginx-auth-request-module/ (visited on 2022-11-26).

[30] Mengyi Li et al. "A Multi-protocol Authentication Shibboleth Framework and Implementation for Identity Federation". In: *Security and Privacy in Communication Networks*. Ed. by Raheem Beyah et al. Cham: Springer International Publishing, 2018, pp. 81–101. ISBN: 978-3-030-01704-0.

[31] Chengwei Liu et al. "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 672–684. ISBN: 9781450392211. DOI: 10.1145/3510003.3510142. URL: https://doi.org/10.1145/3510003.3510142.

[32] Collabora Ltd. *Collabora Online*. 2021. URL: https://www.collaboraoffice.com/collabora-online-3/ (visited on 2022-10-10).

[33] Connect2id Ltd. *LDAP user authentication explained*. 2022. URL: https://connect2id.com/products/ldapauth/auth-explained (visited on 2022-11-18).

[34] D. M'Raihi et al. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. RFC Editor, May 2011, pp. 1–16. URL: https://www.rfc-editor.org/rfc/rfc6238.

[35] Ami Marowka. "Python accelerators for high-performance computing". In: *The Journal of Supercomputing* 74 (Apr. 2018), pp. 1449–1460. DOI: 10.1007/s11227-017-2213-5.

[36] *Matrix*. 2021. URL: https://matrix.org (visited on 2021-12-27).

[37] Clement Michaud. *authelia*. 2022. URL: https://www.authelia.com/ (visited on 2022-01-17).

[38] David Molnar and Stuart Schechter. "Self Hosting vs. Cloud Hosting: Accounting for the security impact of hosting in the cloud". In: *The Ninth Workshop on the Economics of Information Security (WEIS 2010)*. Microsoft Research. 2010. URL: https://econinfosec.org/archive/weis2010/papers/session5/weis2010_schechter.pdf.

[39] Nextcloud. *Your cloud, your rules*. 2021. URL: https://nextcloud.com/athome/ (visited on 2021-12-17).

[40] *Node.js*. 2021. URL: https://nodejs.org/en/about/ (visited on 2021-01-13).

[41] *OAuth Working Group Specifications*. 2022. URL: https://oauth.net/specs/ (visited on 2022-11-16).

[42] Marc Ohm et al. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Clémentine Maurice et al. Cham: Springer International Publishing, 2020, pp. 23–43. ISBN: 978-3-030-52683-2.

[43] Oracle. *Package java.security*. 2023. URL: https://docs.oracle.com/javase/8/docs/api/java/security/package-summary.html (visited on 2023-01-27).

[44] Aaron Pecki. *OAuth 2.0 Simplified*. Lulu Press Inc, 2018. ISBN: 9781387813650.

[45] *PHP*. 2021. URL: https://www.php.net/ (visited on 2022-01-13).

[46] Postfix. *Postfix feature overview*. 2022. URL: http://www.postfix.org/features.html (visited on 2022-01-14).

[47] Michael Pradel and Koushik Sen. "The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 519–541. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.519. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5236.

[48] L. Prechelt. "An empirical comparison of seven programming languages". In: *Computer* 33.10 (2000), pp. 23–29. DOI: 10.1109/2.876288.

[49] William Pugh. "The Java memory model is fatally flawed". In: *Concurrency: Practice and Experience* 12.6 (2000), pp. 445–455. DOI: https://doi.org/10.1002/1096-9128(200005)12:6⟨445::AID-CPE484⟩3.0.CO;2-A. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1096-9128%28200005%2912%3A6%3C445%3A%3AAID-CPE484%3E3.0.CO%3B2-A. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1096-

9128%28200005%2912%3A6%3C445%3A%3AAID-CPE484%3E3.0.CO%3B2-A.

[50] *Python*. 2022. URL: https://www.python.org/ (visited on 2022-01-13).

[51] Ken Reese et al. "A Usability Study of Five Two-Factor Authentication Methods". In: *SOUPS'19: Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*. Ed. by Heather Richter Lipford. Brigham Young University. USENIX Association, Aug. 2019. URL: https://www.usenix.org/system/files/soups2019-reese.pdf.

[52] *Registry — Elixir v1.14.2*. 2022. URL: https://hexdocs.pm/elixir/Registry.html (visited on 2023-01-09).

[53] N. Sakimura et al. *OpenID Connect Core 1.0 incorporating errata set 1*. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 2022-11-17).

[54] N. Sakimura et al. *OpenID Connect Discovery 1.0 incorporating errata set 1*. 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html (visited on 2022-11-18).

[55] Jone Samra. "Comparing Performance of Plain PHP and Four of Its Popular Frameworks". Bachelor's Thesis. Linnaeus University, Department of Computer Science, Aug. 2015. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A846121&dswid=1611.

[56] *Servarr*. 2021. URL: https://wiki.servarr.com/ (visited on 2021-12-17).

[57] Toyotaro Suzumura et al. "Performance Comparison of Web Service Engines in PHP, Java and C". In: *2008 IEEE International Conference on Web Services*. 2008, pp. 385–392. DOI: 10.1109/ICWS.2008.71.

[58] Sylvestre. *sun-java6 packages removed soon from Debian/Ubuntu (and all other linux distros)*. 2011. URL: https://sylvestre.ledru.info/blog/2011/08/26/sun_java6_packages_removed_from_debian_u (visited on 2023-01-27).

[59] The Elixir Team. *Elixir*. 2022. URL: https://elixir-lang.org/ (visited on 2023-01-05).

[60] Debian Wiki. *Java*. 2022. URL: https://wiki.debian.org/Java (visited on 2023-01-05).

[61] Jonas Winkler. *Paperless-ng*. 2021. URL: https://github.com/jonaswinkler/paperless-ng (visited on 2021-12-27).

[62] Kara R. Wong. "Vendor-hosted versus Self-hosted Implementation of Open-Source". MA thesis. Chapel Hill, North Carolina: University of North Carolina, Apr. 2022. URL: https://cdr.lib.unc.edu/downloads/q237j2611.

[63] Yunohost. *Be the cloud you want to see in the world*. 2022. URL: https://yunohost.org/ (visited on 2022-03-10).

[64] K Zeilinga and OpenLDAP Foundation. *LDAP Authentication Password Schema*. RFC 3112. RFC Editor, May 2001, pp. 1–9. URL: https://www.rfc-editor.org/rfc/rfc3112.

[65] K Zeilinga and OpenLDAP Foundation. *Lightweight Directory Access Protocol (LDAP): Directory Information Models*. RFC 4512. RFC Editor, June 2006, pp. 1–52. URL: https://www.rfc-editor.org/rfc/rfc4512.