Wire Federation

version 0.0.4

Wire Swiss GmbH

May 08, 2024

Contents

Wire Federation

Wire Federation aims to allow multiple Wire-server backends to federate with each other: Users on on different backends are be able to interact with each other as if they are on the the same backend.

Federated backends are be able to identify, discover and authenticate one-another using the domain names under which they are reachable via the network. To enable federation, administrators of a Wire backend can decide to either specifically list the backends that they want to federate with, or to allow federation with all Wire backends reachable from the network. See **`configure-federation`**.

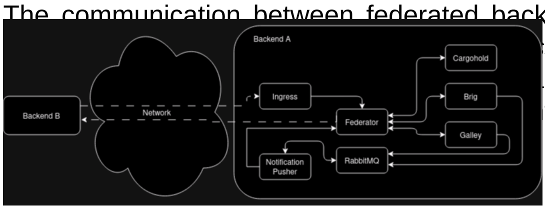
> **Note**
>
> The Federation development is work in progress.

Federation Achitecture

Backends

In the following we call a **backend** the set of servers, databases and DNS configurations that together form one single Wire Server entity as seen from the outside. It can also be called a Wire “instance” or “server” or “Wire installation”. Every resource (e.g. users, conversations, assets and teams) exists and is *owned* by a single backend, which we can refer to as that resource’s backend.

The communication between federated backends is facilitated by two components in each backend: Federation
s is, as the name suggests, the ingress point for incoming connections
rded to the *Federator*. The *Federator* forwards requests to internal
t for requests from internal backend components to other, remote



Backend domains

Each backend has two domain: an infrastructure domain and a backend domain.

The **infrastructure domain** is the domain name under which the backend is actually reachable via the network. It is also the domain name that each backend uses in authenticating itself to other backends.

Similarly, there is the **backend domain**, which is used to qualify the names and identifiers of users local to an individual backend in the context of federation.

The distinction between the two domains allows the owner of a backend domain, e.g. `example.com`, to host their Wire backend under a different infrastructure domain, e.g. `wire.infra.example.com`.

Federation Ingress

The *Federation Ingress* is a Kubernetes ingress and uses nginx as its underlying software.

It is configured with a set of X.509 certificates, which acts as root of trust for the authentication of the infrastructure domain of remote backends, as well as with a certificate, which it uses to authenticate itself toward other backends.

Its functions are:

- to terminate TLS connections
- to perform mutual Authentication as part of the TLS connection establishment
- to forward requests to the local Federator instance, along with the remote backend’s client certificate

Federator

The *Federator* performs additional authorization checks after receiving federated requests from the *Federation Ingress* and acts as egress point for other backend components. It can be configured to use an allow list to authorize incoming and outgoing connections, and it keeps an X.509 client certificate for the backend's infrastructure domain to authenticate itself towards other backends. Additionally, it requires a connection to a DNS resolver to discover other backends.

When receiving a request from an internal component, the *Federator* will:

1. If enabled, ensure the target domain is in the allow list,
2. Discover the other backend,
3. Establish a mutually authenticated channel to the other backend using its client certificate,
4. Send the request to the other backend and
5. Forward the response back to the originating component (and eventually to the originating Wire client).

The *Federator* also implements the authorization logic for incoming requests and acts as intermediary between the *Federation Ingress* and the internal components. The *Federator* will, for incoming requests from remote backends (forwarded via the local Federation Ingress):

1. Discover the mapping between backend domain claimed by the remote backend and its infra domain,
2. Verify that the discovered infrastructure domain matches the domain in the remote backend's client certificate,
3. If enabled, ensure that the backend domain of the other backend is in the allow list.
4. Forward requests to other wire-server components.

Service components

Components such as Brig, Galley, Cargohold are responsible for actual business logic and interfacing with databases and non-federation related external services. See source code documentation. In the context of federation, their functions include:

- For incoming requests from other backends: per-request authorization
- Outgoing requests to other backends are always sent via a local Federator instance.

For more information of the functionalities provided to remote backends through their *Federator*, see the federated API documentation.

Backend to backend communication

We require communication between the Federator of one (sending) backend and the Federation Ingress of another (receiving) backend to be both mutually authenticated and authorized. More specifically, both backends need to ensure the following:

- **Authentication**

  Determine the identity (infrastructure domain name) of the other backend.
- **Discovery**

  Ensure that the other backend is authorized to represent the backend domain claimed by the other backend.
- **Authorization**

  Ensure that this backend is authorized to federate with the other backend.

Authentication

Authentication between Wire backends is achieved using the mutual authentication feature of TLS as defined in RFC 8556.

In particular, this means that the ingress of each backend needs to be provisioned with one or more trusted root certificates to authenticate certificates provided by other backends when accepting incoming connections.

Conversely, every *Federator* needs to be provisioned with a client certificate which it uses to authenticate itself towards other backends.

Note that the client certificate is required to be issued with the backend's infrastructure domain as one of the subject alternative names (SAN), which is defined in RFC 5280.

See **`federation-certificate-setup`** for technical instructions.

If a receiving backend fails to authenticate the client certificate, it fails the request with an `AuthenticationFailure` error.

Discovery

The discovery process allows a backend to determine the infrastructure domain of a given backend domain.

This step is necessary in two scenarios:

- A backend would like to establish a connection to another backend that it only knows the backend domain of. This is the case, for example, when a user of a local backend searches for a qualified username, which only includes the backend domain of that user's backend.
- When receiving a message from another backend that authenticates with a given infrastructure domain and claims to represent a given backend domain, a backend would like to ensure the backend domain owner authorized the owner of the infrastructure domain to run their Wire backend.

To make discovery possible, any party hosting a Wire backend has to announce the infrastructure domain via a DNS *SRV* record as defined in RFC 2782 with `service = wire-server-federator, proto = tcp` and with name pointing to the backend's domain and *target* to the backend's infrastructure domain.

For example, Company A with backend domain *company-a.com* and infrastructure domain *wire.company-a.com* could publish

```
_wire-server-federator._tcp.company-a.com. 600  IN  SRV 10 5 443 federator.wire.company-a.com.
```

A backend can then be discovered, given its domain, by issuing a DNS query for the SRV record specifying the *wire-server-federator* service.

In case this process fails the Federator fails to forward the request with a `DiscoveryFailure` error.

SRV TTL and Caching

After retrieving the SRV record for a given domain, the local backend caches the *backend domain* <–> *infrastructure domain* mapping for the duration indicated in the TTL field of the record.

Due to this caching behavior, the TTL value of the SRV record dictates at which intervals remote backends will refresh their mapping of the local backend's backend domain to infrastructure domain. As a consequence a value in the order of magnitude of 24 hours will reduce the amount of overhead for remote backends.

On the other hand in the setup phase of a backend, or when a change of infrastructure domain is required, a TTL value in the magnitude of a few minutes allows remote backends to recover more quickly from a change of the infrastructure domain.

Authorization

After an incoming connection is authenticated the backend authorizes the request. It does so by verifying that the backend domain of the sender is contained in the **`domain allow list`**.

Since the request is authenticated only by the infrastructure domain the sending backend is required to add its backend domain as a `Wire-Origin-Domain` header to the request. The receiving backend follows the process described in Discovery and verifies that the discovered infrastructure domain for the backend domain indicated in the `Wire-Origin-Domain` header is one of the Subject Alternative Names contained in the client certificate used to sign the request. If this is not the case, the receiving backend fails the request with a `ValidationError`.

Per-request authorization

In addition to the general authorization step that is performed by the federator when a new, mutually authenticated TLS connection is established, the component processing the request performs an additional, per-request authorization step.

How this step is performed depends on the API endpoint, the contents of the request and the context in which it is made.

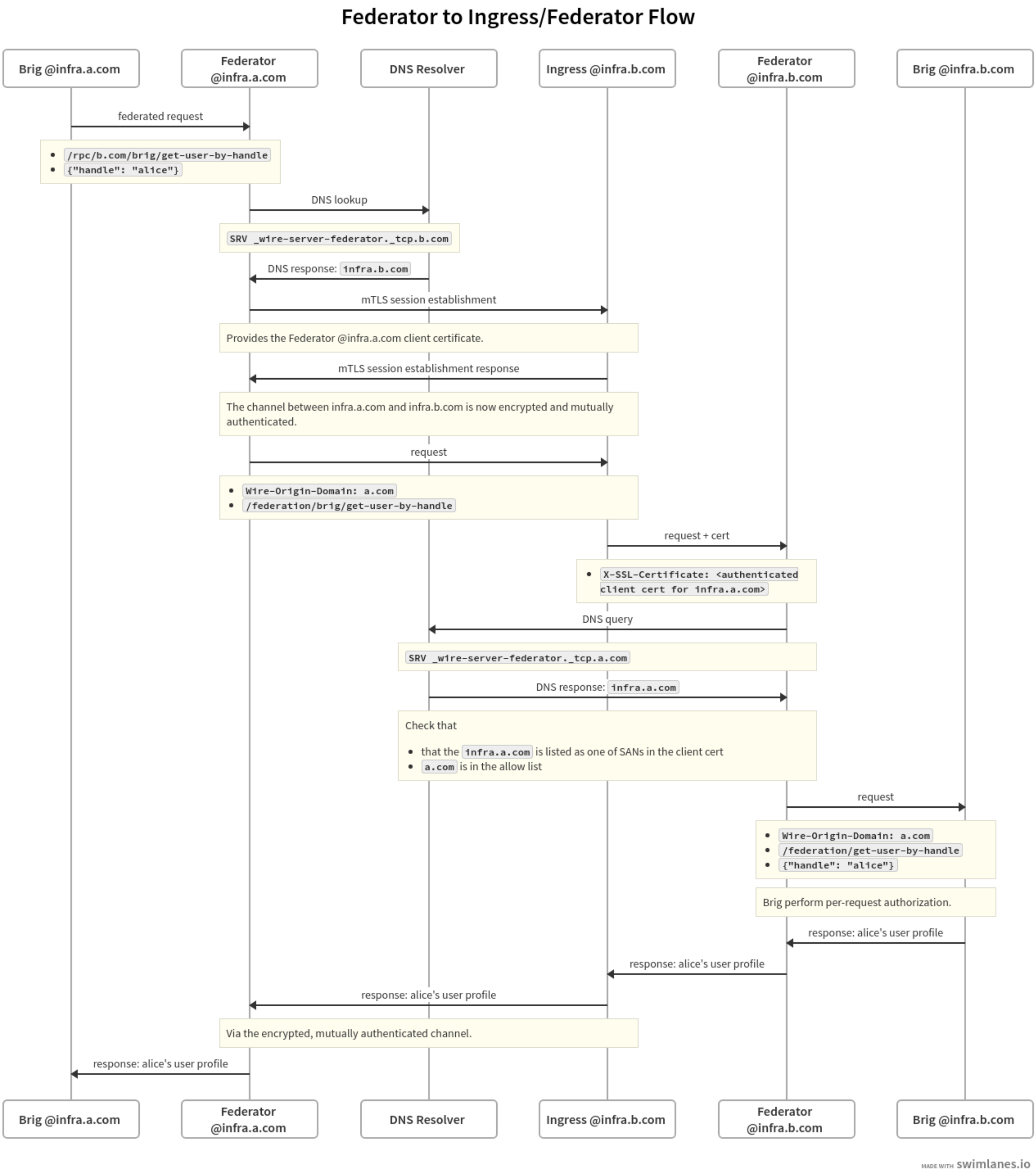
See the documentation of the individual API endpoints for details.

Example

The following is an example for the message and information flow between a backend with backend domain `a.com` and infrastructure domain `infra.a.com` and another backend with backend domain `b.com` and infrastructure domain `infra.b.com`.

The content and format of the message is meant to be representative. For the definitions of the actual payloads, please see the federation API section.

The scenario is that the brig at `infra.a.com` has received a user search request from *Alice*, one of its clients.



Federation API

Qualified Identifiers and Names

The federated architecture is reflected in the structure of the various identifiers and names used in the API. Identifiers, such as user ids, are unique within the context of a backend. They are made unique within the context of all federating backend by combining them with the backend domain.

For example a user with user id `d389b370-5f7d-4efd-9f9a-8d525540ad93` on backend `b.example.com` has the *qualified user id* `d389b370-5f7d-4efd-9f9a-8d525540ad93@b.example.com`. In API request bodies qualified identities are encoded as objects, e.g.

```
{
  "user": {
      "id": "d389b370-5f7d-4efd-9f9a-8d525540ad93",
      "domain": "b.example.com"
  }
  ...
}
```

In API path segments qualified identities are encoded with the domain first, e.g.

```
POST /connections/b.example.com/d389b370-5f7d-4efd-9f9a-8d525540ad93
```

to send a connection request to a user.

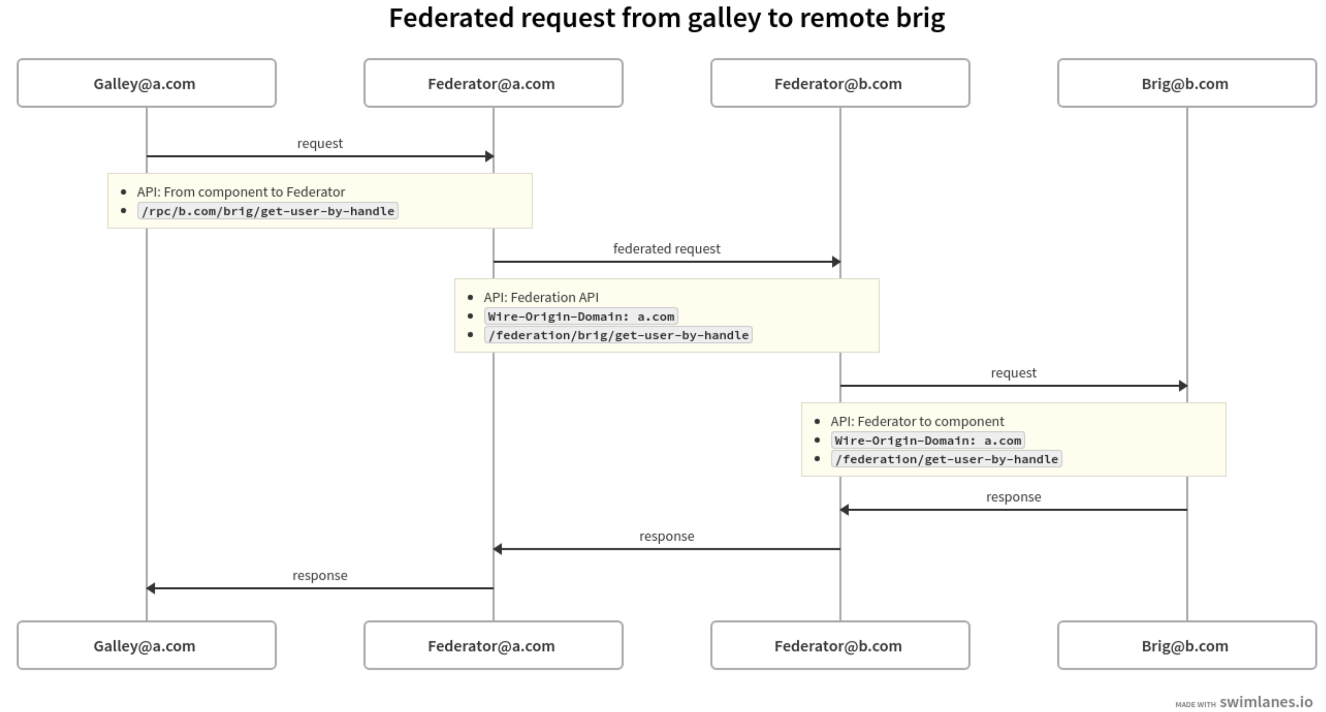
Any identifier on a backend can be qualified:

- conversation ids
- team ids
- client ids
- user ids
- user handles, e.g. local handle `@alice` is displayed as `@alice@b.example.com` in federating users' devices

User profile names (e.g. "Alice") which are not unique on the user's backend, can be changed by the user at any time and are not qualified.

Federated requests

Every federated API request is made by a service component (e.g. brig, galley, cargohold) in one backend and responded to by a service component in the other backend. The *Federators* of the backends are relaying the request between the components across backends . The components talk to each other via the *Federator* in the originating domain and *Federator Ingress* in the receiving domain (for details see Backend to backend communication).



Federators relaying a request between components. See Example *to see the discovery, authentication and authorization steps that are omitted from this figure.*

API From Components to Federator

When making the call to the *Federator*, the components use HTTP2. They call the Federator's `Outward` service, which accepts `POST` requests with path `/rpc/:domain/:component/:rpc`. Such a request will be forwarded to the remote Federator with the given backend domain, and converted to the appropriate request of its `Inward` service.

API between Federators

The layer between *Federator* acts as an envelope for communication between other components of wire server. The *Inward* service of *Federator* is an HTTP2 server which is responsible for accepting external requests coming from other backends, and forwarding them to the appropriate component (currently Brig or Galley).

Federator inspects the header of an incoming requests, performs discovery and authentication, as described in Backend to backend communication, then forwards the request as-is by repackaging its body into an HTTP request for the target component.

The *Inward* service accepts only `POST` requests with a path of the form `/federation/:component/:rpc`, where `:component` is the lowercase name of the target component (i.e. `brig` or `galley`), and `:rpc` is the name of the federation RPC to invoke. The arguments of the RPC are contained the body, which is assumed to be of content type `application/json`.

See API From Federator to Components for more details on RPCs and their paths.

API From Federator to Components

The components expose a REST API over HTTP to be consumed by the *Federator*. All the paths start with `/federation`. When a *Federator* receives a `POST` request to `/federation/brig/get-user-by-handle`, it connects to a local Brig and forwards the request to it after changing its path to `/federation/get-user-by-handle`.

The `/federation` prefix is kept in the path to allow the component to distinguish federated requests from requests by clients or other local components.

If this request succeeds, the response is directly used as a response for the original call to the `Inward` service. Otherwise, a response with a `5xx` status code is returned, with a body containing a description of the error that has occurred.

Note that the name of the RPC (`get-user-by-handle` in the above example) is required to be a single path segment consisting of only ASCII characters within a restricted set. This prevents path-traversal attacks such as attempting to access `/federation/../users/by-handle`.

List of Federation APIs exposed by Components

Each component of the backend provides an API towards the *Federator* for access by other backends.

> **Note**
>
> This reflects status of API endpoints as of 2023-01-10. For latest APIs please refer to the corresponding source code linked in the individual section.

Brig

In its current state, the primary purpose of the Brig API is to allow users of remote backends to create conversations with the local users of the backend.

- `get-user-by-handle`: Given a handle, return the user profile corresponding to that handle.

- `get-users-by-ids`: Given a list of user ids, return the list of corresponding user profiles.
- `claim-prekey`: Given a user id and a client id, return a Proteus pre-key belonging to that user.
- `claim-prekey-bundle`: Given a user id, return a prekey for each of the user's clients.
- `claim-multi-prekey-bundle`: Given a list of user ids, return prekeys of their respective clients.
- `search-users`: Given a term, search the user database for matches w.r.t. that term.
- `get-user-clients`: Given a list of user ids, return the lists of clients of each of the users.
- `get-user-clients`: Given a list of user ids, return a list of all their clients with public information
- `send-connection-action`: Make and also respond to user connection requests
- `on-user-deleted-connections`: Notify users that are connected to remote user about that user's deletion
- `get-mls-clients`: Request all **`MLS`**-capable clients for a given user
- `claim-key-packages`: Claim a previously-uploaded KeyPackage of a remote user. User for adding users to MLS conversations.

See the brig source code for the current list of federated endpoints of *Brig*, as well as their precise inputs and outputs.

Galley

Each backend keeps a record of the conversations that each of its members is a part of. The purpose of the Galley API is to allow backends to synchronize the state of the conversations of their members.

- `get-conversations`: Given a qualified user id and a list of conversation ids, return the details of the conversations. This allows a remote backend to query conversation metadata of their local user from this backend. To avoid metadata leaks, the backend will check that the domain of the given user corresponds to the domain of the backend sending the request.
- `get-sub-conversation`: Get a MLS subconversation
- `leave-conversation`: Given a remote user and a conversation id, remove the the remote user from the (local) conversation.
- `mls-welcome`: Send MLS welcome message to a new user owned by the called backend
- `on-client-removed`: Inform called backend that a client of a user has been deleted
- `on-conversation-created`: Given a name and a list of conversation members, create a conversation locally. This is used to inform another backend of a new conversation that involves their local user(s).
- `on-conversation-updated`: Given a qualified user id and a qualified conversation id, update the conversation details locally with the other data provided. This is used to alert remote backend of updates in the conversation metadata of conversations in which at least one of their local users is involved.
- `on-message-sent`: Given a remote message and a conversation id, propagate a message to local users. This is used whenever there is a remote user in a conversation (see end-to-end flows).
- `on-mls-message-sent`: Receive a MLS message that originates in the calling backend
- `update-typing-indicator`: Used by the calling backend (that does not own the conversation) to inform the backend about a change of the typing indicator status of one of its users
- `on-typing-indicator-updated`: Used by the calling backend (that owns a conversation) to inform the called backend about a change of the typing indicator status of remote user
- `on-user-deleted-conversations`: When a user on calling backend this request is made for all conversations on the called backend was part of
- `query-group-info`: Query the MLS public group state
- `send-message`: Given a sender and a raw message request, send a message to a conversation owned by another backend. This is used when the user sending a message is not on the same backend as the conversation the message is sent in.
- `send-mls-commit-bundle`: Send a MLS commit bundle to backend that owns the conversation

- `send-mls-message`: Send MLS message to backend that owns the conversation
- `update-conversation`: Calling backend requests a conversation action on the called backend which owns the conversation

See the galley source code for the current list of federated endpoints of *Galley*, as well as their precise inputs and outputs.

Cargohold

- `get-asset`: Check if asset owned by called backend is available to calling backend
- `stream-asset`: Stream asset owned by the called backend

See the cargohold source code for the current list of federated endpoints of the *Cargohold*, as well as their precise inputs and outputs.

Example End-to-End Flows

In the following the interactions between *Federator* and *Federation Ingress* components of the backends involved are omitted for simplicity. Also the backend domain and infrastructure domain are assumed the same.

Additionally we assume that the backend domain and the infrastructure domain of the respective backends involved are the same and each domain identifies a distinct backend.

User Discovery

In this flow, the user *Alice* at *a.example.com* tries to search for user *Bob* at *b.example.com*.

1. User *Alice* enters the qualified user name of the target user *Bob* : `@bob@b.example.com` into the search field of their Wire client.
2. The client issues a query to `/search/contacts` of the Brig searching for *Bob* at *b.example.com*.
3. The Brig in *Alice*'s backend asks its local *Federator* to query the `search-users` endpoint in *Bob*'s backend.
4. *Alice*'s *Federator* queries *Bob*'s Brig via *Bob*'s *Federation Ingress* and *Federator* as requested.
5. *Bob*'s Brig replies with *Bob*'s user name and qualified handle, the response goes through *Bob*'s *Federator* and *Federation Ingress*, as well as *Alice*'s *Federator* before it reaches *A*'s Brig.
6. *Alice*'s Brig forwards that information to *A*'s client.

Conversation Establishment

After having discovered user *Bob* at *b.example.com*, user *Alice* at *a.example.com* wants to establish a conversation with *Bob*.

1. From the search results of a user discovery process, *Alice* chooses to create a conversation with *Bob*.
2. *Alice*'s client issues a `/users/b.example.com/<bobs-user-id>/prekeys` query to *Alice*'s Brig.
3. *Alice*'s Brig asks its *Federator* to query the `claim-prekey-bundle` endpoint of *Bob*'s backend using *Bob*'s user id.
4. *Bob*'s *Federation Ingress* forwards the query to the *Federator*, who in turn forwards it to the local Brig.
5. *Bob*'s Brig replies with a prekey bundle for each of *Bob*'s clients, which is forwarded to *Alice*'s Brig via *Bob*'s *Federator* and *Federation Ingress*, as well as *Alice*'s *Federator*.
6. *Alice*'s Brig forwards that information to *A*'s client.
7. *Alice*'s client queries the `/conversations` endpoint of its Galley using *Bob*'s user id.
8. *Alice*'s Galley creates the conversation locally and queries the `on-conversation-created` endpoint of *Bob*'s Galley (again via its local *Federator*, as well as *Bob*'s *Federation Ingress* and *Federator*) to inform it about the new conversation, including the conversation metadata in the request.
9. *Bob*'s Galley registers the conversation locally and confirms the query.

10. *Bob*'s Galley notifies *Bob*'s client of the creation of the conversation.

Message Sending

Having established a conversation with user *Bob* at *b.example.com*, user *Alice* at *a.example.com* wants to send a message to user *Bob*.

1. In a conversation *<conv-id-1>@a.example.com* on *Alice*'s backend with users *Alice* and *Bob*, *Alice* sends a message by using the `/conversations/a.example.com/<conv-id-1>/proteus/messages` endpoint on *Alice*'s Galley.
2. *Alice*'s Galley checks if *A* included all necessary user devices in their request. For that it makes a `get-user-clients` request to *Bob*'s Galley. *Alice*'s Galley checks that the returned list of clients matches the list of clients the message was encrypted for.
3. *Alice*'s Galley sends the message to all clients in the conversation that are part of *Alice*'s backend.
4. *Alice*'s Galley queries the `on-message-sent` endpoint on *Bob*'s Galley via its *Federator* and *Bob*'s *Federation Ingress* and *Federator*.
5. *Bob*'s Galley will propagate the message to all local clients involved in the conversation.

Ownership

Wire uses the concept of **ownership** as a guiding principle in the design of Federation. Every resource, e.g. user, conversation, asset, is **owned** by the backend on which it was *created*.

A backend that owns a resource is the source of truth for it. For example, for users this means that information about user *Alice* which is owned by backend *A* is stored only on backend *A*. If any federating backend needs information about the user *Alice*, e.g. the profile information, it needs to request that information from *A*.

In some cases backends locally store partial information of resources they don't own. For example a backend stores a reference to any remotely-owned conversation any of its users is participating in. However, to get the full list of all participants of a remote conversation, the owning backend needs to be queried.

Ownership is reflected in the naming convention of federation RPCs. Any rpc named with prefix `on-` is always invoked by the backend that owns the resource to inform federating backends. For example, if a user leaves a remote conversation its backend would call the `leave-conversation` rpc on the remote conversation. The remote backend would remove the user and inform all other federating backends that participate in that conversation of this change by calling their `on-conversation-updated` rpc.

Federated API calls by client API end-point (generated)

Updated manually using using the fedcalls tool; last change: 2023-01-16.

This is most likely only interesting for backend developers.

This graph and csv file describe which public (client) API end-points trigger calls to which end-points at backends federating with the one that is called. The data is correct by construction (see the fedcalls tool for more details).

The target can only be understood in the context of the backend code base. It is described by component (sub-directory in `/services`) and end-point name (use grep to find it).

links:

- **`dot`**
- **`png`**
- **`csv`**

Federated API calls by client API end-point (generated)