integrity.

Distributed trust for the access economy

White paper

Patrik Wallström, with
Jonas Bosson
Remy Bourganel
Erik Hedenström
Lars Larsson

The +Integrity infrastructure simplifies trusted communication and avoids risks inherent to the current trend of more centralized data. +Integrity offers a decentralized trust infrastructure that enables validated identities, secure transactions and signatures, in a decentralized service model. An organisation can set up a security realm and issue mandates to users for use with its services. Users receive receipts when using these services that may be used to communicate trust with other actors on the +Integrity Platform.

Note, the terminology used in this document may differ from that which a user may see in an application based on the technology described.

support@plusintegrity.com
plusintegrity.com
Github, Glitter

1. Background

There are many weaknesses with centralized identities and data storage systems. The number of data breaches has not decreased over the years, and although there is intense interest in protecting passwords and sensitive data, by technical security measures and regulatory means, there is nothing in sight stopping centralized data sources from future data breaches.

Besides all efforts to protect data in centralized services, there are many alternative decentralized solutions now proven to be equally or more secure than conventional systems. With the advent and wide-scale use of blockchains, as popularised by Bitcoin, distributed hash tables, as popularised by BitTorrent, anonymising networking technologies such as Tor and I2P, and distributed file system technologies such as IPFS and Tahoe-LAFS, there is ample evidence that building secure decentralized systems work in practice. These distributed systems have laid the foundation for a new decentralized network infrastructure that can scale and better support the web.

Ubiquitous computing platforms, such as the ‘smart’ class of mobile phones combined with new web technologies and universal internet access is already in the public’s hands. This threshold has been passed some years ago. This increasingly capable eco-system where new distributed platforms need trust, identities and documents to be distributed without the inefficiencies of centralized storage is already there. As consumers’ awareness of the privacy implications of using centralized cloud services increases, the latent capacity inherent in this infrastructure is a huge opportunity. By transferring data previously locked into centralized services to this new paradigm it provides a lower friction, higher integrity solution to connect consumers with businesses and services.

1.1. A new infrastructure for distributed identities and data

+Integrity is a new decentralized infrastructure for managing identities, performing transactions, signing documents, agreements, and performing actions. Actions execute within the context of autonomous organisational security domains called Realms.
A *Realm* is an independent organisational structure that secures and manages users and applications. Realms grant access through *Roles* and *Mandates* giving users the power to perform actions and the ability to discover more services.

Services are provided by a *Controller* and controllers are only loosely coupled to the Realm, in that they are bound only by a cryptographic bootstrap where each party has mutual trust through their public keys. In reality the two can be in totally separate topologies in the network.

In order for the user to perform actions on a controller, the user needs to first to discover the service. This is achieved in multiple ways, all a user needs is to receive a document or a reference to a document describing the service. The broadcasting of *Action Descriptors* (or the details of a service) can be done with e-mails, web links, tags on a door or machine or through local network discovery mechanisms such as mDNS.

Some centralized services will be available, if a realm chooses to make use of them. Those services, for example might be a listing of services available within a realm, KYC services and an *Identity Provider* IdP exposing the OAuth2 or SAML methods for authorizing users. Anybody can host the necessary central services as part of another +Integrity infrastructure setup. Users and/or the realm provider can choose which parts of the infrastructure setup to trust.

A realm can also choose to trust other Realms - if the organisational Realm wants to trust KYC or IdP services from an infrastructure realm, that trust can be delegated. The same goes for trust of 3rd party Realms, they exchange public keys and their users can move between Realms. Using Facts issued by an infrastructure Realm bestows upon its users the freedom to join other Realms and use their services without any need to perform a new KYC. A user can also choose to exchange receipts with a new Realm as proof of the user's identity.

1. 2. The user in control

The identity of a user on the +Integrity Platform is composed by data stored only in the users phone protected by the +Integrity App. When a user first installs the app and creates a new identity, a new assymetric key pair is generated. The public part of this key is the root User ID (UID), and identifies users throughout the system. However, subkeys can be generated for specific purposes and additional privacy.

A public user ID is usually not enough for somebody else to trust this user. Many services require some facts that identify the user, such as e-mail addresses, phone numbers, a known social media identity, passport or national eID. A Fact may also consist of ephemeral data such as a geographic location. To create trusted Facts, a KYC ("Know Your Customer") service offered by a trusted Realm can be used by the customer to collect a number of facts about the user. These facts are then stored in the user's phone, managed by the user. When a service or Realm requests facts as part of a transaction these facts can be shared, but only if the user allows them to be shared. If the user has many facts of the same type, such as multiple e-mail addresses, the user selects which e-mail address to share with the service.

In order to perform an *Action* for a service, the user has to have *Mandates*, granting the user the authority to perform certain actions. Mandates are provided by Realms offering a service. An *Action* contains a signed action document that also wraps the mandate - the mandate is signed by the Realm and the action is signed by the user. In this way, the service controller can validate the signatures over the Action and Mandate, and follow the signatures back to the issuing Realm. If the Action(s) is validated correctly according to the service policy, the Action is executed.

Actions produce *Receipts* that are stored in the user's logbook. Along with a receipt from performing an action the user could also receive a number of new mandates unlocking access to other services. These receipts could be signed agreements, room bookings, or facts of ownership with links to related services.

A transaction performed in this way is only seen by the controller and the user - no other party is capable of decoding the transaction.

The benefits to the user are many, primarily in how the user can control how to share their data, and that any transaction only contains the information strictly necessary for the transaction to be performed. An additional example, this will simplify the enrollment process for users connecting to new services, as it will remove the repetition of exchanging paperwork, filling in and signing multiple forms to receive keys and getting access to wifi - and other the other cumbersome steps to store and maintain these permissions, passwords, codes and items that are generated in response. Consider a classic onboarding case. When a user has been through a signup process for a service a few times, the user will have enough facts collected that can easily be shared with a new service to validate them. In these two example aspects the system simplifies and enriches the services and workflows for the interacting parties by exploiting previous activity and digital capturing the process output into a secure user controlled service.

1. 3. Easy integration with 3rd party services

The +Integrity Identity Provider (IdP) is compatible with OAuth2 and SAML, using a +Integrity identity to authenticate with external services. This means that +Integrity users may log into a vast array of corporate and public services with their +Integrity identity.

The +Integrity mobile phone app also offers pluggable controls and brandable realms, providing an easy path to integrate branded services and experiences into the +Integrity App. The pluggable services can wrap external services by using our crypto library. Adding a administration and user interface to the service is done with standard tooling, such as HTML.

1. 4. Distributed identities, no centralized data

Data associated with a user (in essence, the users public key, facts and receipts) is stored in the phone, and encrypted in a distributed storage system such as IPFS. This removes any senstive data storage liability for a service as this data is not stored by the service, only logs and receipts that may be a result of an action. A user can login to services connected to our open IdP, using OAuth or SAML, using the Integrity app. The +Integrity IdP service does not store any information at all besides other than the public keys of other realms.

Given that users has control over their data, this also means that users also have full responsibility to protect their data and keys. In order to help users with this responsibility, there must be tools available to make secure backups. The most convenient method to store a backup is to encrypt it and store it in one place. However, by using “secret sharing” (or “secret splitting”) users may backup partial copies of their data or keys encrypted to the public key of their friends. Restoring the key when re-installing the app and the phone can then be done by asking their friends to send back their part of the key. Options for storing secrets using PBKDF2 or Argon2 with a strong password will also be available.

New privacy regulations such as GDPR makes protecting personal data in storage, transit and while processing, essential for companies in order to limit their liability. De-coupling personal data from centralized systems will allow people to share personal data with less friction and greater trust supporting economic activity.

1. 5. The remote control to your environment

The +Integrity App can be used as a remote control for access to services around you. Our focus is to de-mystify and simplify the complex interactions that ensure the security, reliability of users’ data. All whilst ensuring the scalability of the +Integrity infrastructure.

With automatic discovery of services and geographical discovery of Realms, it will be easy for the user to reach the services they need. If there is a tag on a door or a meeting room, or any object, the App quickly creates the correct Action to interact with the object.

If there are many nearby Realms for the user to choose from, the +Integrity App makes it easy to switch between the Realms based on previous interactions. As long as you have a relationship with the Realm, the services they offer are very easy to access.

The requirements for joining a Realm might differ a lot. Some Realms might require some facts to be shared, some may ask for an agreement to be signed, and some Realms might even be completely open and offer memberships just to offer their services. In all those cases, the signup process is very brief, all that is required is to sign any agreement and share the facts needed, and you will receive an appropriate mandate. Many realms may offer a “guest” mandate that gives the user a set of limited services to access, but enough to see what the realm can offer.

A controller can offer just about any type of service - a marketplace for the exchange of objects, an interaction with an IoT device, or a vending machine. To initiate the interaction of a service the +Integrity App needs access to the Action Descriptor of the service. The descriptor can be a tag the user scans, or part of what the realm offers through its service listing. If there is a UIURL in the descriptor, the app opens that URL when the user wants to use the service. This URL can be hosted anywhere, but the look and feel is the same as in the rest of the app. The user will never notice that external services are “external” other than the user interface will take a somewhat longer time to open. Once the interaction with the UI of a controller is done (all parameters that needs to be able to perform an action has been collected), the app creates an Action, signs it and sends it to the controller endpoint and performs the action.

So what is needed to create a good user experience for any new services, +Integrity will offer libraries for developers to create controllers and offer user interface guidelines to streamline the user experience.

2. Introduction to +Integrity

+Integrity will be a complete infrastructure for building a decentralized identity system with no centralized storage. The main user interface for the users in this system is the +Integrity App, and the users identity is stored in a secure enclave in the phone. The app is central in this system, and it acts as the users remote control for using the services offered by the different realms. However, through clever use of tokens and subkeys, websites accessible through a desktop browser can be used as well.

A +Integrity infrastructure realm is trusted by default in the app, and this realm offers essential functionality such as a KYC service, an Identity Provider that offers third party integration for authenticating users (SAML and OAuth), and a discovery service where the user can discover other realms that offer publically available services.

To bridge the IPFS storage the infrastructure realm also provides a storage API that also provides the reverse linking necessary for finding document links.

A realm does not need the +Integrity infrastructure realm to offer full functionality to a user, the organisation that runs a realm can choose to run the necessary services themselves. Or they can trust any other realm to do it for them, and thus create a chain of trust to other parties. A standalone realm can be very useful if the user just wants to access personal services e.g. in a home.

All services offered on the +Integrity Platform are published through controllers as action descriptors. The descriptors are picked up by the app and converted into actions and sent to the controllers to perform the action. If more parameters are needed to perform an action, an optional UI can be presented by the controller. The user is sent to the UI to populate the action with the missing parameters, and all parameters from the users mandate and from a form in the UI are baked into an action document that the controller parses and verifies.

There are many types of document handled by the app, and a transaction can generate a multipart document consisting of several other documents, for example a receipt of an action and a message.

2. 1. From a users perspective

Respecting the user is primary in the +Integrity infrastructure. A user should never be asked to disclose any information not necessary for using a service. Once the data is disclosed to a third party (such as a service), the user control over the data is lost. Regulations and data protection laws puts constraints to what services can do with user data. Although the +Integrity Platform is designed to secure communications between systems and protect user data, there is no silver bullet to ensure that there is no data leakage.

2. 1. 1. Signup

A person can become a +Integrity user for a number of different reasons. The most probable scenario is an invite from someone, asking them to join a meeting or giving them access to a service. The invitation will contain a link for the meeting, which will refer the user to install or use the +Integrity App.

When starting the +Integrity App for the first time it will brifly introduce the user to the concepts of the app. A new identity for the user will be created (the user root key), unless the user wants to retrieve the identity from any backup.

If the app was triggered by an external action, as giving the user a Mandate, the Mandate will be added to the list of the users Mandates. However, if the link triggers the app store to ask the user to install the app for the first time, the user has to go through the signup process, and then click the link again in order to receive the mandate.

Signing up to a realm, or being handed certain mandates, or performing actions, may require that the user has certain facts collected and also be willing to share them. It is easy from within the app to add new facts, the +Integrity App is using a default infrastructure realm that has a KYC service. Realms and services that has been configured to trust the +Integrity Platform automatically trust facts that are signed with keys that comes from the +Integrity Platform and its KYC service.

The mandate gives the user the ability to act in a specific role, such as Employee, Staff, Guest or VIP. The mandate can also be even more specific, such as giving the user access to only one or few actions for a very explicit purpose. A service may choose to handle the roles different depending on configuration, giving access just to certain roles, or discounts on the cost of different services.

2.1. 2. Discovering services

There are multiple ways for the user to discover services. A Controller may have the capability to announce its services (called Action Descriptors) on the local network as mDNS broadcasts, the administrator of the Realm can creats tags (QR codes, barcodes, NFC tags) and physically put them close to the service, or links to the services on a web page, or through bots in a Slack channel.

There are also discovery mechanisms offered by the infrastructure realm where registered realms can publish their services. When the app queries the discovery service it can base queries on interests or geographic proximity.

A realm itself can offer a service listing. To avoid abuse, the service listing may only be available to users with an established relation to the realm, where the user might have at least a “guest” mandate. If the user presents a better mandate (or even an administrators mandate), the service listing may return a complete list of services available to both users and administrators.

Some services are only available on request, and might not be discoverable at all. These services can send their Action Descriptor through any channel available to the user.

2. 1. 3. Joining a realm

Joining a realm may not always mean anything other than a means to discover services, using a “guest” role. The meanings of different roles in a realm may differ a lot between different realms, as well as the meaning of “being a member” of a realm.

For a user to join a new realm the requirements for joining may also differ a lot. Some realms might require some facts to be shared, some may ask for an agreement to be signed, and some

realms might even be completely open and offer memberships in order to just be able to advertise or broadcast their services. In all those cases, the signup process is very brief, all that is required is to sign any agreement and share any required facts, and you will receive one or more appropriate mandates. Many realms may offer a “guest” mandate that gives the user a set of limited services to access, but enough to discover what other services the realm can offer.

In many cases, joining a realm starts with that a user wants to do something with a service. If the user does not have a relation to the realm (lacking a mandate) that the service is bound to, there may be an optional mandate request URL in the action descriptor. When the +Integrity App discovers that it lacks a matching mandate, it can asks the user to follow any mandate request URL. This URL can contain a web page with a signup form for the user. When having completed the signup the user can get a matching mandate and start to use the service.

2.1.4. Use services

To use a service offered by a realm you need a mandate. The mandate does not mean much if no mapping for it exists in the configuration the controller offering the service. A service offered by a controller is broadcasted through mDNS, through tags or any channel provided by the realm (e-mail, web sites etc.), and can be discovered by the +Integrity App. Then it will show up under services in the realm in the app, or listed as nearby services. The service is published as an Action Descriptor or a referral (URL) to an Action Descriptor.

To use the service the user uses the +Integrity App, or the app might be triggered by opening an Action Descriptor URL (from e.g. a web application or opened from an e-mail). The app parses the Action Descriptor and looks for the ActionURI and the UIURI. If there is no UIURI the app configures an Action document and posts it to the ActionURI to perform the action. If there is an UIURI, the app opens the web view published there. This URI contains (if everything is setup correctly) a web view containing an interface to configure parameters to perform the Action. When the user is done with the UIURI web page, the Action is configured and posted to the ActionURI.

Common results from an action is either a message (ok, or perhaps an error), or a receipt containing detailed information about the performed Action. It can also contain the information about future times if it was the result of a booking of a resource. In reality, the result from an action can be just about anything, and contain multiple documents with multiple document types (such as Mandates or signed Facts).

2.2. From a service providers perspective

In order to offer new types of services in a +Integrity realm, you have to create a controller. A controller is the software that provides the services to users. As a service provider you may already have a set of APIs to access the functionality offered by your service. A controller can just map any actions coming from users in the +Integrity infrastructure to API calls in the provider system.

To create a new controller, what is needed is the crypto library provided from +Integrity. The functions needed in the controller is an endpoint for the Action Descriptor, and an endpoint for performing actions. To have a nice user experience and make it easy for realm administrators to set up and configure the controller, an administration interface is also needed. Also, if any extra parameters are needed in order to perform the Action, a user interface for inputting that data is also helpful.

In order to bind the controller with one or more realms, you need endpoints to help this binding as well. This will make it easy for a realm administrator to configure the service.

New services can be added to a marketplace for controllers, where realm administrators can find and discover controllers for their realms. Services may not be free of use, and there will be mechanisms in place to facilitate any transactions needed to purchase controllers. Third party services to be used by a controller are outside of the +Integrity Platform.

2. 3. Setting up a Realm

A realm in the +Integrity infrastructure is a very thin layer of functions that manages relations in the form of mandates, and connecting services. Optionally the realm can trust data coming from other realms, used for example by trusting an external KYC service for facts coming from users.

A realm announces itself with a Realm Descriptor. The Realm Descriptor is a signed JSON document that is published in the .well-known directory with HTTPS for the domain name used to host the realm. The descriptor contains the public key(s) to the realm, and this key is what users and services is going to put trust in.

On first use after installing the Realm software, or setting up the Realm in a hosted environment, the person initiating the Realm will receive a unique token (or URL) that will make the person the administrator (or owner) of the realm. This user will have their key associated with a user object in the realm. The administrator then proceed to set up further roles in the realm, such as those used by administrators, staff or guests (although the use of roles are very flexible in this regard).

In order to begin to bind controller to the realm, the Realm Descriptor must be published in the .well-known directory, so that the controller can read the keys from the realm. However, once the realm software has been installed, and the administrator has been added, and the .well-known file is in place, all is set for use of the realm. The administrator can log in and manage roles, mandates and controllers.

2.3.1. Adding services

Without services (controllers) a realm is quite useless. So the first task after setting up the realm is to add a service.

The kind of services the realm will use will depend very much of what purpose the realm has. There will be a marketplace for ready-made services, already hosted by others. The realm administrator will pick and choose from these services.

Other services may be integration with local devices, and for more specialized controllers source code (or even pre-compiled binaries) may be available freely for installation on the local network. In that case, connecting the controller to the realm is going to be just as easy as installing from the marketplace, with the exception of installing the software itself.

After installing the controller and connecting it to the realm, it has to be configured for use by users with other roles. Configuration is done in the administration interface of the controller, and this is also where roles are designated to what users can do with the controller.

More exact details of the binding between a controller and a realm is outlines in the chapter “Binding a Controller to a Realm”.

2.3.2. Managing roles and users

Users are not handled by realms per se on the +Integrity Platform. How a user relates to a realm is by receiving a mandate that gives the user a power to act in a certain role. What roles mean in a specific realm context is up to the realm, and how the controllers bound to the realm are configured to handle roles. A controller can handle services only to very specific roles such as “room1-booking@realm.example”, or very broad defined roles such as “employee@realm.example”. Also, a controller may be configured to handle roles in any way that fits its purpose, and any role can have a whole range of limitations, such as what time of day the role can perform any actions - it depends on the functionality of the controller.

The realm can list the current set of roles used in the realm, and when issuing mandates this mandate can be limited in the number of uses and that it is only issued for one role. A mandate can also be limited in time, when an expire date for the mandate is set.

Even though a mandate has been issued to a user, a user may not be able to use it with a service. The service may require that the user presents certain facts in order to use the service. This can be for example a phone number, an e-mail address, or just about anything. Not until the user can present the required facts for a controller, any action on behalf of the user using the mandate can be performed.

2.3.3. Issuing Mandates

Through the realm user interface you can create a mandate for a certain role. The mandate can have an expiration date (TTL) that tells the user and controllers that the mandate is no longer valid after a certain point, which allows for temporary mandates to be issued.

The realm can chose to publish (share) the mandate through the web site and give the URL to any user. In reality, what the user gets is a Scope Request, meaning that the user must present certain required facts in order to receive the mandate. This could be for example an e-mail address, or even an e-mail address for a certain domain (belonging to a specific organisation). When the user has shared the required facts, the app will receive the mandate.

A mandate can also be issued directly from the realm to a user and sent in an e-mail, or in whatever form the app can read (as a tag for example). A controller can also issue a mandate, if it has a proper subkey for issuing mandates (of a certain type) on behalf of a realm.

2.4. Use cases

There are many practical use cases for an identity federation system like this, however we would like to higlight a few of them. Except for the obvious applications such as logging in users to services, there are a number of other practical applications one can consider.

2.4.1. Enable booking of conference rooms

An office wants to enable people without direct access to their current calendar system to book their conference rooms. They set up a realm for their office and binds the +Integrity booking controller to their realm, and then connect their calendar to the booking controller. Now, the office manager can choose can connect calendars for the resources that ny member of the realm can book through the booking service.

Any user that wants to book the resource (conference room in this case) can scan the tag published on the door of the room, and if the user lacks the Integrity app the phone will open the app store with the page for the Integrity app.

2.4.2. Access to internal only services

An office might also want to share some services that are only shared within the local network (LAN). The controllers can announce their services with mDNS, and users can easily be enabled to use services only exposed on the LAN through those controllers with the Integrity app. The office only have to share the right mandates to the users. Any such controller does not need internet access, except for checking for revoked keys.

2.4.3. Document signing

It’s easy to write controllers that asks the user to sign documents via the app as most actions on the +Integrity Platform create small signed documents or messages.

An example is signing the acceptance of end-user license agreement for becoming a member of a realm.

A more complex example is a service where a document has to be signed by several different parties. This could be a specialized controller that follows a strict procedure for a specialized document signing for a specific purpose. The controller can send out an e-mail or use another messaging service to notify all parties and ask them to sign the document(s), and when all parties har signed the documents, the process proceeds with finalizing the document and perhaps performs the next step in the process (e.g. sending out a signed copies of what has been signed).

3. Technical overview

The three major components of the +Integrity infrastructure consists of the realm software, the +Integrity App, and controllers that the user interacts with to use services.

1. The realm software in its core is a management interface to manage mandates for the users. It is also used to bind controllers to the realm and publish them in a directory listing. And finally, it publishes it public key, which is the trust anchor for users interacting with controllers.

2. The mobile app Integrity is the hub in which the user manages the personal identity and the different documents around the identity (receipts, facts and mandates). In a simplified view the app is a document handler, since it can send and receive +Integrity documents of the different types - and sign, encrypt and validate them.

3. Controllers are what the user will mainly interact with when using the Integrity app, and use services. Services can be discovered outside of the app, and upon opening a URL that is bound with the app it will trigger the action to use the service of a controller. Controllers are recieving signed actions from a user to perform its services.

The three different components are bound together by cryptographic trust and not by closely connected API calls. A controller does not need a direct connection to a realm to perform services on behalf of the user, the user rather needs a cryptographic trust to make use of services in the realm context.

3.1. +Integrity App - The mobile component

The mobile app +Integrity App is the remote control for every use of the +Integrity Platform. This is where the users keys are stored (“My profile”), and it is where the receipts, facts, mandates and eveything else that belongs to the user are stored.

There are views for service discovery, setting context depending on nearby realms, functions for scanning any +Integrity document (mandates, action descriptors and so on), performing actions, and managing the users log (receipts and such) and documents.

If you are the administrator of a realm or controller you can use these mandates to enter the administration pages of these things. However, none of this functionality resides within the app, but is loaded as web views in the app. This will happen without the user noticing that the views are loaded from another host, and if the controller is designed using the design standards the look and feel of the views will be the same as for the rest of the Integrity app.

When performing an action, any UI needed to configure the action (adding missing parameters, such as dates needed for a booking service) will also be loaded from the controller. If the receipt from an action contains a reference to a UI, the receipt can be rendered with a custom view from the controller as well. This is useful if the user wants to cancel or change the details afterwards.

3.1.1. Document handling

The +Integrity App registers itself as a link handler for any link that begins with https://plusintegrity.com/. This means that if any such link is opened on the mobile device it will be handled by the app. If the integrity app is not installed the link would cause a browser to open and it would load a page that encourages the user to install the +Integrity App.

As an alternative to the deep linking method, a user can also trigger the app to handle links by scanning a QR code or NFC tag. The code or tag can contain a link to a document to be handled, or the entire document itself.

When the app has been passed a link or document via deep linking or QR/NFC- scanning it passes it to its document handler module. The document handler will attempt to fetch a document if it was given a URI. Once it has a document it will decrypt it if it is encrypted, and any signatures will be validated. Next it will pass the document into the correct handler based upon the document type. The result from the handler will be returned to the app or view that initiated the link or scan.

3. 1. 2. Web views / 3rd Party UI

When performing an action by handling an action descriptor it is sometimes neccessary to collect additional parameters. For example when booking a conference room a user must be given the option of selecting what time and which duration they need the room for. The +Integrity App uses webviews to provide a generic solution for configuring actions.

When handling an action descriptor, if it contains a a UI URI it will open a webview and load the URI. Optionally the UI data can be inlined in the action descriptor and set statically in the webview. The webview is sandboxed and has a very limited interface to the native functions of the app.

A loaded webview has the following lifecycle:

1. Webview is instantiated and given the UI URI to load
2. Webview has loaded the URI and is made visible to the user
3. The app calls the init function in the webview passing in the mandate and params from the action descriptor.
4. The app polls the pollfunction to see if the the webview is done.
   - If the poll function returns a value the webview is closed and the result is used to complete the action.
   - If a cancel flag is returned the webview is closed and the action is cancelled.
   - If a null value is returned the app continues the next step.
5. The app polls the handle function to see if there are documents that need to be handled by native functions.If a document is returned it is passed to the document handler. The result of the document handler is passed back to the webview via the result or error functions.
6. After a small delay the app repeats the polling process.

3.1.3. Discovery

Discovery of realms and services can happen in several ways. The user can tap a link (coming from anywhere, even chat or e-mail messages) on their device to trigger the Integrity app. Alternatively the user scan a QR code or NFC tag to activate a link. The app can also receive push notifications that trigger the document handler.

Public local realms and services can be discovered through a geographical lookup. The app can perform a query with it's location as parameters and receive back a list of nearby realms and their services, and presented by relevance to the user.

Private local realms and services can be discovered by mDNS broadcasts on the same wifi network that the device is attached to.

3.1.4. Key handling

When the +Integrity App is run the first time it generates a root key. The private part of the key is protected by a user defined PIN code. The public part of the root key is signed and then published to IPFS. The returned IPFS document id is a SHA-256 of the contents encoded in base58, is set as the key ID. This ID is what uniquely identifies the user throughout the +Integrity Platform.

When the public key has been successfully published and stored the app continues to generate an app keypair. The app key is signed by the root key and protected by the same pin code as the user root key. The app key is used for document signing and decryption, while the root key is only used for signing new user keys and for decryption of documents.

A third key called device is also generated and signed by the user root key. It is not protected by any pin code. It is used for document signing and decryption when performing low value operations such as registering a device for push notifications. Some actions may be flagged as low value operations and this allow a pinless action that can be performed in the background, for example, booking a conference room simply by scanning an NFC tag.

All the keys generated by the app are stored in the secure storage of the device. Exactly how the secure storage is implemented is device specific. On iOS the data is stored in the system Keychain. On Android it uses the Keystore which may be hardware or software based depending upon the device.

3. 1. 5. Facts, Mandates, and Receipts

When performing actions or handling documents certain returned documents are stored by the app. Facts are signed verifiable claims. Mandates entitle users to perform actions. And receipts are the results of such actions. Each of these documents are stored in a simple SQL database. The user can browse and manage facts and mandates via their profile page in the app.

The receipts can trigger webviews if a URI is specified. This enables users to make changes to previous actions by performing new actions via the webview. For example, cancelling a room booking by tapping on the receipt.

3.1.6. Logbook - past and future events

All receipts received by the app are presented in a log. Receipts are listed in chronological order based upon the time at which the action is realised for the user. For example room booking receipts are timestamped upon creation but presented in the logbook at the time which the room is booked. The intent of this listing model is to make it easy for the user to find and manage current and future events.

3.2. Distributed storage

The +Integrity infrastructure makes use of two different technologies for distributed storage. Storing immutable files is done with IPFS, which is a distributed file storage system. Such files can be retrieved by any other IPFS node connected to the internet. Blockchains are used for key revocation, where a service that must verify a transaction is performed with valid keys can do lookups without leaking any data to a third part.

Ideally, any blockchain or IPFS node should be run as close to the clients and servers using them as possible. Sometimes they are exposed by an API, thus making the API endpoint a possible source for data leakage. However, in this distributed model, any party can run an API endpoint since they are all using the same storage.

3.2.1. Privacy vs publishing

Most of the data in +Integrity is only relevant to the two parties that communicates, the user and a realm with its services, and there is no need to publish or communicate the documents through other parties.

There some are cases where there is a need to publish information in a way so it is accessible by anyone. Such information could be:

- Key revocations
- Public profiles and keys
- Signatures of certain document types

3.2.2. IPFS

“IPFS (the InterPlanetary File System) is a new hypermedia distribution protocol, addressed by content and identities.”

IPFS is a system for distributed content-addressable (meaning the hash of the data becomes the address) storage. As such all data in IPFS is immutable. When data objects are stored in IPFS a hash is calculated of the data and that becomes the address where the data can be found. The IPFS address can be used by anyone in the IPFS network to find your data, as long as there is at least one copy of the data online in the network. Data is not replicated and distributed to other nodes automatically, another node in the IPFS network has to query for the object in order for their local node to fetch the data and cache it. When the +Integrity infrastructure storage system is used it makes sure that the data is replicated to multiple nodes.

+Integrity uses IPFS to publish the public key of +Integrity users and realms in order to create a globally unique address for each key.

3.2.3. Linked documents

+Integrity documents are based JSON Linked Data (JSON-LD) that provides the basic structure for how to link documents to other related documents.

An example use of this is a user that needs to collect signatures on a document. The document can be published as a Signature Request document in IPFS, share the document address, and then ask users to sign the document. The users would create Signature documents that would link back to your original Signature Request. This makes it is easy to follow the path through the documents, and see what the related documents refer to.

3.2.4. Reverse links

Since the underlying storage is using IPFS to set unique addresses for stored documents, and IPFS is content addressable, it is impossible to know the address before the document is final. If there are any changes to the document the document address be changed as well.

This is a problem if you want to link to something that happens after you have stored your document in IPFS. The example in the previous section described linking to already existing documents, but what if you would like to find all Signature documents that is linking to your Signature Request document?
The solution to this is to have a separate metadata service that parses the links in documents, and adds them to the metadata service as a reverse link. A client can query the metadata service for documents that may have a newer version, or a chain of documents referring back to older documents.

3.2.5. Publishing

When data should be published to the entire network we use IPFS storage combined with links and reverse links, and package multiple such operations in to a single document that we publish on a blockchain. We are agnostic about which blockchain to use as long as there is a consensus in one or many networks.

3.3. The Anatomy of a Realm

A Realm is an independent organisational structure that manages users and services. What that means in practice is that an organisation typically sets up one realm for its users, whether those are employees or members are not important as the users are defined by their roles and mandates they receive.

In essence a realm is a web based application with a backend with a simpler storage for its configuration. As long as the realm can list controllers for services and issue mandates, and issue a realm descriptor, it is usable with the +Integrity App.

A Realm defines a set of Roles that can be used througout the realm. It is up to the administrator of the Realm to define what the roles are going to be used for. Typical roles might be Employee, Guest, Staff or Concierge, it depends on what type of Realm and what services it might offer.

Once some roles has been defined, the realm can also issue Mandates. The mandates gives the user to capability to act in a certain Role. An mandate can be limited in time or the number of uses

3.4. Controllers and services

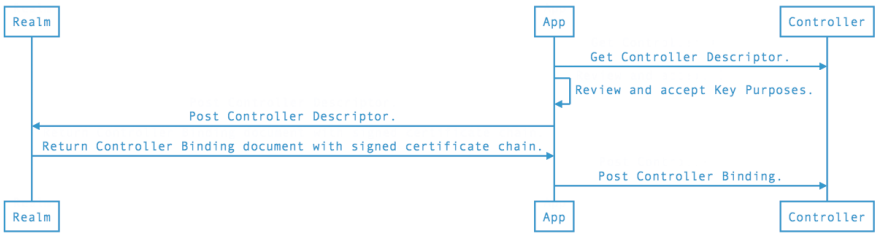
A realm exposes an API that a controller uses for binding its services, and some metadata about them such as the API endpoints for the controller. The realm can also list the current set of services (controllers) and their endpoints to any client asking for them, as a discovery mechanism.
There is no guarantee that a controller can communicate over the network directly with a realm or vice versa, since these two functions can be separated by networks that prohibits direct connections. So all communications between the controller and the realm is done through the Integrity app (or the web interface). When an admin manages the controller belonging to the realm, the admin device acts as the proxy for all communication necessary to make any change of configuration in either the realm or the controller.

3.4.1. Binding a Controller to a Realm

The controller will initially be functional enough to present a *Controller Descriptor.* A descriptor contains the name of the controller (Label), an ActionsURI, an AdminURI, a BindURI and the key, and key purposes.

The Controller Descriptor is used by the realm to initialize a binding with the controller. Using the Integrity app, the realm gets the Controller Descriptor, and posts back (again through the app) a Controller Binding document to the controller.

>



The *Controller* Binding document contains the Realm Descriptor (same as the .well-known document published by the realm), the list of Admin Roles that are allowed to manage this controller, a certificate chain that is a signed proof that the controller key is allowed to sign documents on behalf of the realm, and a service mandate that is used by the controller to interact with the realm or other services. (Consider the case of meta-services, where a controller can in turn trigger several other actions.)

When the binding is done, the controller can perform services on behalf of the realm. However, to be fully functional, the controller should probably be configured to use other roles without permissions to manage the controller, and connect to any third party service it might make use of. So the last step is for the realm administrator to enter the administration interface of the controller, and perform necessary configuration.

A controller can be bound to many realms, in which case the procedure above have to be repeated for every realm.

3.4.2. Controller Admin interface Single-Sign-On

The Realm administration interface has a collection of links to the controller administration interfaces.

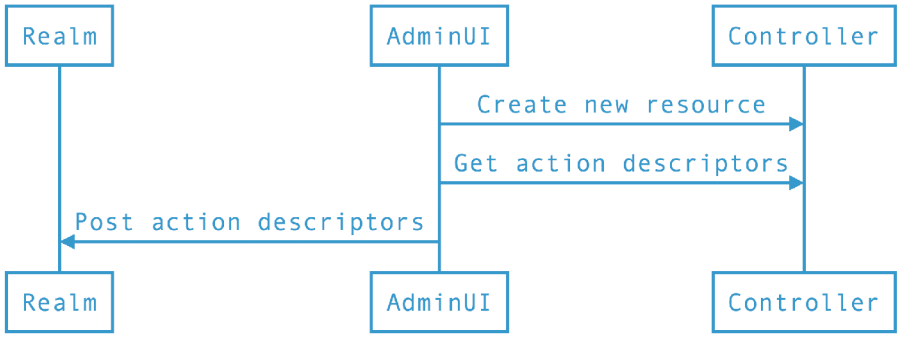
When following one of those links an SSO token is generated in the realm, and that token can be verified by the controller and automatically login the user.

The token also contains a API token that the controller administration interface can use to make limited API calls to the realm for operations such as looking up roles and publishing action descriptors.

3.4.3. Publishing action descriptors from a controller

The administration interface of the controller will publish the Action Descriptors to the realm after having added new Actions or modified existing ones.

>



3.5. Meta services

From simpler controllers that performs Actions such as unlocking a lock and booking a resource you can construct more complex services. An example of this would be a booking service for meetings where you want to book a room, send out invitations to the meeting, hand out a key to let people let themselves in, and order food and other services.

To accomplish this more complex task you can construct a controller that binds to the needed services as a meta service. This meta service has a controller that inherrits Mandates to create Actions on other Controllers.

3.6. Keys, trust and storage.

All keys on the +Integrity Platform are JSON Web Keys (JWK), and all signature are JSON Web Signatures (JWS). JSON Web Keys has been produced through a standardization effort in the IETF (JOSE, JSON Object Signing and Encryption) and published as a series of RFCs (most notably RFC7515). +Integrity currently uses ECDSA P-256 and SHA-256 for cryptographic operations. Using a standard framework such as JOSE makes it easy to be cryptographically agile when any algorithm shows any signs of weakness.
Any signature, key, or document, that a +Integrity App user wants to validate (successfully) has to be signed by a party that the app already trusts (initially the +Integrity infastructure realm key). The full chain of signatures must be validate successfully to accept any incoming signature. By publish a new realm through the +Integrity infrastructure makes it implicit that any new +Integrity App user can trust any documents issued by the new realm. Any user that encounters a new realm that has not been published (or signed) through the +Integrity infrastructure can validate the realm key from the .well-known file from the domain it belongs to.

In the +Integrity App, the personal private keys (including any subkeys) are stored in the Secure Enclave of the device. The exact security of this key storage may vary among the vendors and the versions of the operating systems, but most new devices have a very secure key storage partition. Signing any document in the app requires the user to input pin (or fingerprint) in order to use the keys stored in the device.

Key backup can be done in multiple ways, either using paper keys for storing a backup completely offline, or encrypted using a strong password and PBKDF2 (or bcrypt, possibly Argon2) in a storage considered safe by the user. Using the users friends it should also be possible to construct a split key and share it among some trusted friends using Shamir's Secret Sharing. When the secure backup of the key has been done, a backup of the rest of the encrypted user data in a decentralized storage can be done, using the users public key.

A private key can be compromised or lost, or for some reason revealed by mistake. These keys should not be trusted from the point they are identified in a key revokation list. +Integrity wants to avoid having a centralized key revocation service. Since all important transactions need to check for revoked keys, all controllers that checks for a revoked key will reveal some metadata for the keys involved in the transaction to a central revokation service. So what we are doing is publishing key revocations in a blockchain. This will make the key irreversebly revoked. Controllers that needs to do key revocation lookups can use a local blockchain to lookup keys. This avoids the issue of leaking metadata from a controller.

3.7. IDP and KYC services

A +Integrity identity can be used to log into other services by using a +Integrity IDP service. The advantage of an IDP service from +Integrity over other similar services is that the IDP service will not contain any user information such as names, e-mail addresses, or any passwords.

A +Integrity IDP is configured to trust one or more realms (with KYC services) and their public keys, and not much more than that. This makes it a lot less vulnerable to huge data leaks than centralized services, which is an enormous benefit to anybody that operates the service.

Already implemented is OpenID Connect which is a simple identity layer of the OAuth 2.0 protocol. Any Relying Party (RP, typically an external service provider) that wants to use a +Integrity IDP as their authentication system can choose to run the IDP themself, or pick any existing IDP and add it to their login system.

A Relying Party can decide what Facts are needed to accept from the +Integrity IDP login. A Fact is something that the user has collected as proof of possession of e-mail addresses, phone numbers, etc. Facts can also be for different social media accounts (Facebook, Twitter, Github etc.), or ephemeral facts such as a self-signed geographical position. Stronger identity facts such as the verification of a passport, or a national id card or drivers license is also possible for a KYC service to sign as Facts for the user.

Depending on the amount of trust a service has on a KYC service, and the need for a high level assurance, the service may require that facts are provided by a KYC service that issues facts with a very high level of certainty. Some KYC services can have processes where the person is identified with a very high level of certainty. A service may also require several facts from different trusted issuers to build a sufficient level of assurance.

4. Document types

All document types used to communicate within the +Integrity infrastructure is composed by JSON-based (RFC 7159) data structures with JSON Web Signatures (RFC 7515) (JWS), and encryption using JSON Web Encryption (RFC 7516) (JWE).

All +Integrity document types are based on the +Integrity Base document type.

```
type Base struct {
        Context          string
        Type             string
        SubType          string
        Timestamp        time.Time
        ID               string
        Links            []Link
        Owners           []string
        Callbacks        []string
        CertificateChain string
        Realm            string
        mu               *sync.Mutex
}
```


The Context is in the +Integrity context always set to “https://plusintegrity.com/schema”. The type of the document is the name of the type.

4.1. Realm Descriptor

A Realm Descriptor is typically announced through a domain name, placed in the .well-known directory of a HTTP server. The realm descriptor contains a general description and properties of the Realm, but most importantly it contains the public key identifying the Realm.

```
type RealmDescriptor struct {
        Base
        Name         string
        PublicKey    *jose.JsonWebKey
        InviteURL    string
        ServicesURL  string
        KeyHistory   []string
}
```

The KeyHistory is a list of previous keys used by the realm. The Realm Descriptor document itself is signed by the current and the previous key. By also signing with the previous key, this makes it very easy for a realm to roll keys without any service distruption.

4.2. Facts

A *Fact* can be issued by any actor on the +Integrity Platform, even issued by the user itself. The user can issue a self-signed Fact stating his or her name or the geographical location of the device.

Facts are shared by the user to controllers when a service require them for certain actions. With the +Integrity App a user can choose to share the facts, and which facts to share. In order to receive a mandate from a realm, the user may be required to share facts with the realm as well.

```
type Fact struct {
        Base
        TTL       time.Duration
        Issuer    string
        Label     string
        Data      map[string]interface{}
        Recipient *jose.JsonWebKey
}
```

The user are being asked to share facts when they receive a ScopeRequest, which might happen when a user wants to perform an Action. The ScopeRequest includes a list of the wanted scopes (i.e. which facts are needed) to get from the user.

```
type ScopeRequest struct {
        Base
        ReplyTo   []string
        Scopes    []Scope
        EncryptTo []string
}
```

4.3. Mandates

A Mandate is the document that gives the +Integrity users the capability to perform Actions in the +Integrity system. A mandate itself however does not contain any information about which services it can be used for. The Role is the important field in this document.

```
type Mandate struct {
        Base
        Role          string
        Label         string
        TTL           int
        Recipient     string
        RecipientName string
        RecipientPK   *jose.JsonWebKey
        RequestID     string
        Sender        string
        Params        map[string]string
}
```


4.4. Action and Action Descriptor

An Action Descriptor is typically published by the realm and its controllers. Publishing of the descriptor can be done in any way possible, through mDNS or through a tag - when the Integrity app is triggered by an Action Descriptor the user is typically redirected to the ActionURI. Or if a UIURI is in the descriptor, the app opens this URI in a web view so that the user can configure the action. If the Action Descriptor is already populated with all parameters needed for the Action to be triggered, the UI might redirect the app directly to the controller and perform the Action immediately.

If the service need any facts from the user to allow the usage of a service, the list of scopes is needed to put in the *Action* document as a list of *Facts*.

```
type ActionDescriptor struct {
        Base
        Binding   string
        Label     string
        Roles     []string
        UIURI     string
        UIData    string
        NonceType string
        Nonce     string
        NonceURI  string
        ActionURI string
        Params    map[string]string
        Scopes    []Scope
        Icon      string
}
```

In order to avoid replay attacks, the controller can publish a nonce that the app needs to retrieve before the action is posted - the nonce needs to be included in the action.

When the app and the UI has collected all the parameters needed to perform the Action, the Action document is signed by the user and posted to the Controller.

```
type Action struct {
        Base
        Role    string
        Mandate string
        Nonce   string
        Params  map[string]string
        Facts   []Part
}
```

Depending on the result of the Action, the user receives or more documents, typically a Message or a Receipt. If more than one document is sent by the Controller it will be a Multipart documen

4.5. Messages and Receipts

A Message is a simple document typically sent to a user to display the result of an Action. A message is used if there is no need to give the user a receipt, for example if there is an error or a non-important action.

```
type Message struct {
        Base
        Title   string
        Message string
}
```

A Receipt is a bit more complex, since it also includes the Action (the JSON encoded original document that triggered the Action), any intervals (future dates) that the Receipt is valid for. The intervals are also used to represent the Receipt as future events in the users logbook in the app.

```
type Receipt struct {
        Base
        Role      string
        Action    string
        URI       string
        JWT       string
        Intervals []Interval
        Label     string
}
```


The UI in the Receipt can be used by the app to present a nicer layout of the content of the Receipt, but it could also be possible to update for example a calendar entry.

4.6. Controller Descriptor and Controller Binding

The Controller Descriptor describes the Controller and list the URIs for the API endpoints. The BindURI is the URI of what address the Realm should talk to when creating the binding between the Realm and the Controller, using a “Controller Binding” document.

The AdminURI is the administration web interface for the controller, and the ActionsURI is the address to get a list of Action Descriptors created by this Controller.

The Label and the Key is the controllers name and the public key is published key.

Key Purposes is information on what permissions the Controllers key wants. This will be added to the certificate chain.

```
type ControllerDescriptor struct {
        Base
        Label       string
        ActionsURI  string
        AdminUI     string
        BindURI     string
        Key         *jose.JsonWebKey
        KeyPurposes []KeyPurpose
}
```

The Controller Descriptor is used by the Realm to initialize a binding with the Controller. The Realm posts back the Controller Binding document to the controller.

```
type type ControllerBinding struct {
        Base
        RealmDescriptor            *RealmDescriptor
        AdminRoles                 []string
        ControllerCertificateChain string
        Mandate                    string
}
```

The Controller Binding document contains the Realm Descriptor document (which is the same as the well-known document published by the Realm), the list of Admin Roles, and a certificate chain that is the signed proof that allows the controller to sign other documents on behalf of the realm, and a service mandate that is used by the controller to interact with the realm or other controllers.

5. Conclusion

With the bulding blocks of Realms, Controllers and the Integrity app, we believe we have achieved a fully working distributed identity system with services that respectfully treats the personal information of the user, and gives the user control of their personal information.

Using controllers separate from the realm organisational structure we build trust between systems based on cryptograhic certificate chains. This results in a system where every compontent can act autonomously, without any central authority, where we can respect the privacy of users.