Softwareprojekt 12 “Guardian”
Architecture Fine Design Document

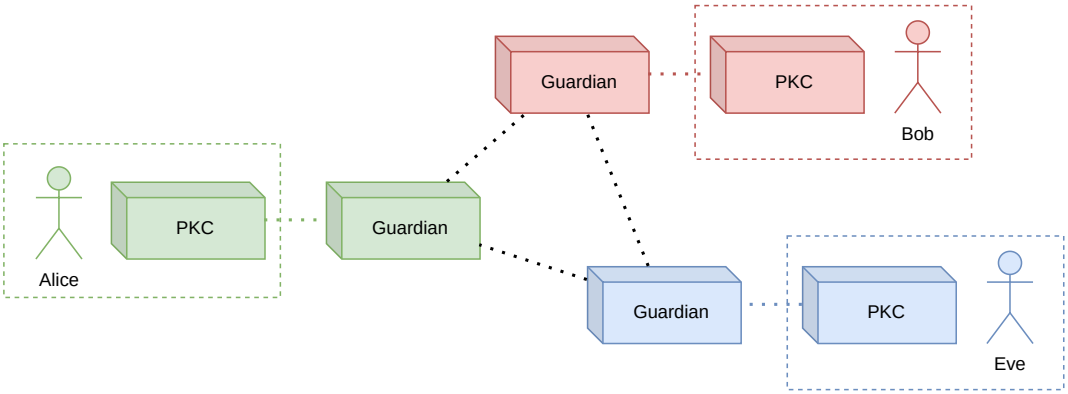
October 16, 2024

Project:	Softwareprojekt 12 “Guardian”
Client:	inblock.io assets GmbH
Contractor:	Ilmenau University of Technology

Version	**Date**	**Author(s)**
1.4	2024.06.14	Mika Elias Richter, Tim Bansemer, Ruben Hans Ehritt

Contents

1 Introduction



This document describes the architecture and design of the Guardian. It is a security component which acts as an enforcement point with two major motivations:

1. Protection of Aqua Containers (PKC) against unauthorized external access
2. Connecting Aqua Containers for File-Exchange

Incoming and outgoing Aqua-Chains are verified before passing through. Invalid Aqua-Chains are rejected. Aqua-Contracts control the permissioned data exchange performed by the Guardian. This allows to define verifiable policies within an Aqua-Chains structured as an Aqua-Contract to represent a policy which is enforced by the Guardian if all validity conditions are met.

The Guardian extends the existing Aqua ecosystem with automated and regulated sharing of data. The current ecosystem consists of:

- The PKC is an Aqua Container implementation to create and manage Aqua-Chains.
- The Verifier Webextension and a command line implementation of the Verification functionality.

2 Design Documentation

This section documents contracts (special types of Aqua-Chain), trust interactions (usage of contracts to build trust), `guardian` system components and their functionalities and APIs the Guardian interacts with.

2.1 Contracts

Contracts within the project context are Aqua-Chains with a defined data structure and validity conditions. Contracts are only put into effect when signed. Their meaning is contextual and depends on the type of contract.

For practical reasons the contract structure is oriented to the MediaWiki template functionality. This gives users a form-based UI for instantiating and configuring templates, resulting in a human readable document.

2.1.1 Guardian Identity Claim

The Guardian Identity Claim proves the existence of a Guardian and shows which account it is serving.

Properties: `guardian_account`, `authoritative_user_account`

Validity Conditions:

- valid signature from `guardian_account`

2.1.2 Guardian API definitions

The Guardian API definitions declare a Guardian's means of communication to other guardians. They are issued by the Guardians themselves for other Guardians to connect to them securely.

Properties: `guardian_identity`, `tls_certificate`, `host`, `port`, `protocols`

Validity Conditions:

- `guardian_identity` hash refers to a valid Identity Claim
- the `tls_certificate` is still valid (did not expire)
- valid signature from the `guardian_account` from the `guardian_identity`

2.1.3 Data Access Agreement

Data Access Agreements control the sharing of Aqua-Chains which allow making files public (sharing them with all trusted Guardians) or permissioned (sharing them with selected Guardians).

Permissioned access is provided with contractual policies (sharing them with a single user that has agreed to the data usage conditions forming an "agreement"). If policies are set both parties have to sign the contract to put it into effect.

Properties: `sender`, `receiver`, `files`, `terms`

Validity Conditions:

- valid signature from `sender`
- if any `terms` are present, a second revision exists with unchanged content signed exclusively by `receiver`

2.2 Trust-based interactions

Since the Guardian's objective is to facilitate secure data exchange, trust-based interactions between Guardian and PKC as well as between different Guardians are required.

2.2.1 Guardian to Guardian Trust

Guardians are using mTLS to have secure connections. The trust comes by the mutually imported certificates.

- Guardian Servitude: It is a contract which is read from the PKC. The Guardian Servitude is issued and signed by the Guardian and needs to be signed be the user to take effect.
- TLS Certificate: It is issued by the Guardian called "TlsCert" which is an Aqua-Contract. The TLS certificate is issued and signed by the Guardian which is read from the PKC.

Those certificates are checked by the receiving parties and are manually imported.

An implementation into an Aqua-Contract structure which automatically verifies and imports the certificates is a future development goal.

See more about the mTLS handling in the appendix.

2.2.2 Managing permissioned access

As listed in the Specifiction Book we want to realise the use-case of sharing Aqua-Chains. The user manages access permissions and policies with a Data Access Agreement.

For this to work, we already must have established trusted communication between Guardians.

Sequence for initiating Data Access:

1. User (blue) instantiates a Data Access Agreement from template (1).
2. User (blue) adds remote user account who should receive files (2.1)
3. User (blue) adds his own account as account of the sender (3)
4. User (blue) adds reference to the Aqua-Chain (Revision Hash) (4).
5. User (blue) signs Data Access Agreement (5).
6. the Guardian (blue) requests Aqua-Chain (6).

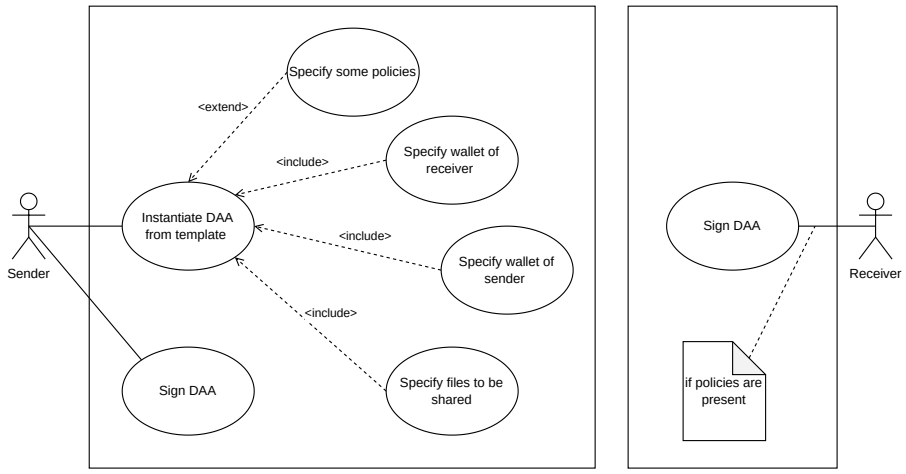


Figure 1: Use-Case for Data Access Agreement

7. the Guardian (blue) verifies the Aqua-Chain and detects Data Access Agreement (7).

   At this point it is considered valid if the following condition is satisfied:

   - Signature is from authoritative user (blue)

8. the remote Guardian (red) requests Aqua-Chains (9).

The Data Access Agreement is either shared with or without policies.

Option (A): NO policies are included

8. the Guardian shares the Aqua-Chain data with the recipient (10).
9. the remote Guardian (red) verifies the incoming Aqua-Chain (11).
10. If Aqua-Chain is valid its imported into the User-Inbox (12).
11. User (red) is notified (13).

Option (B): ANY policies are included:

8. the Guardian (blue) shares the Data Access Agreement without the Aqua-Chain data it defines access to with the recipient Guardian (red) (10).
9. the remote Guardian (red) verifies the incoming Aqua-Chain and detects Data Access Agreement (11).
10. If Aqua-Chain is valid, it is imported into the PKC (red) (12).
11. User (red) is notified (13).
12. The recipient user (red) signs the Agreement (15).
13. Guardian (red) verifies the Data Access Agreement (16).

    The Aqua-Chain needs to be signed by the authoritative user (red).

14. Remote Guardian (red) shares the signed Agreement with Guardian (blue) (17).

15. Guardian (blue) receives and verifies the Data-Access- Agreement (17-18)

    The Aqua-Chain needs to be signed by the authoritative user (red) as well as by the authoritative user (blue).

16. DAA requirements are met and Aqua-Chain data is now sent to remote Guardian (red) as described in (A).

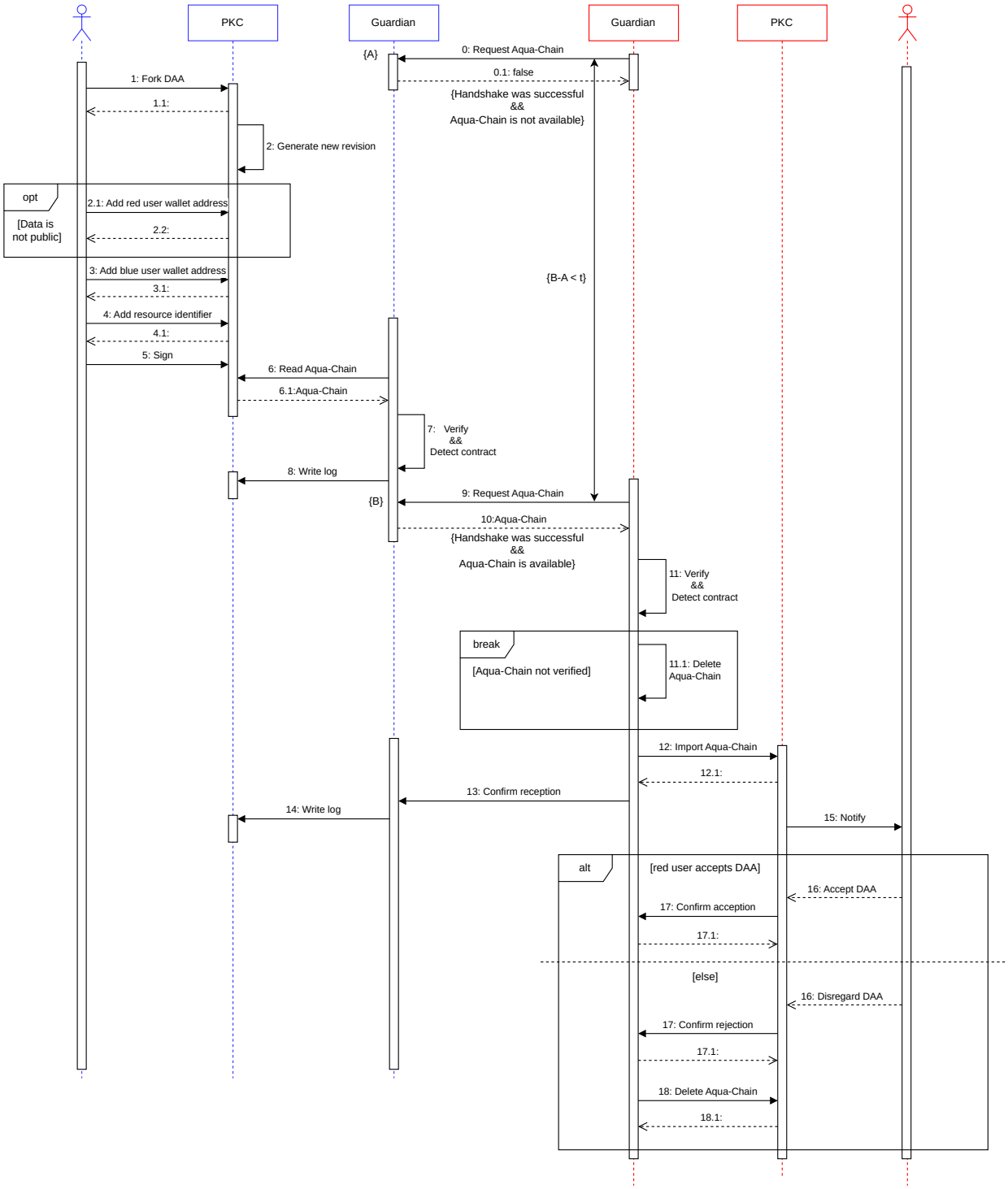
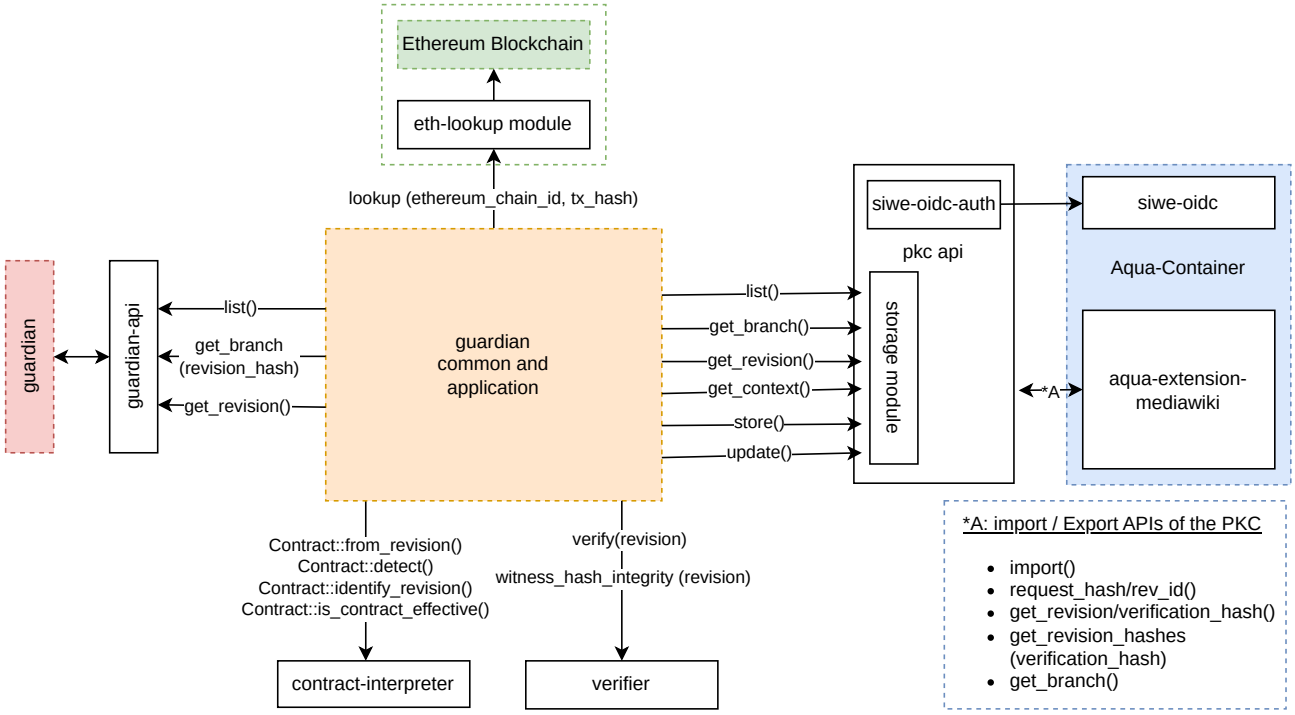


Figure 2: Initiate Data Access

2.3 System Components

The Guardian is separated into several defined components. This section gives the details on which components exist and how they are composed to form the final product.



At the heart of the `guardian` sits the `verifier` which is used to check all incoming Aqua-Chain data. It manages Aqua Containers using a `storage module`.

The `pkc-api` is the `storage module` in this implementation and connects the `guardian` to the PKC.

`siwe-oidc-auth` is a subcomponent of `pkc-api` and is responsible for letting the guardian authorize itself to the PKC.

The `contract-interpreter` detects if an Aqua-Chain is a known contract and parses it if so.

The `guardian-api` component implements the bidirectional communication with other Guardians through the Guardian API.

Part of the verification of Aqua-Chains is also the witness event verification, for this the `verifier` employs an `eth-lookup module`.

2.3.1 `guardian` application

`Guardian` application is an executable which is configured by environment variables. The `guardian` application connects the components internally and handles tasks and events. The asynchronous multi-threaded runtime `tokio` is used to handle concurrent multi-tasking. The component `guardian` application represents the core application and includes and depends on all the other components.

Guardian environment variables:

- "PRIVATE_KEY" Private key of Guardian Ethereum Account
  used to calculate the Public key and wallet address of Guardian Ethereum Account
- "PKC_URL" URL path to the Personal-Knowledge-Container
  example: http://localhost:9352/ to connect the Guardian server
- "ADMIN_USER"
  the wallet address of the authoritative user of the PKC which is given in the PKC setup script
- "INFURA_API_KEY"
  used by the `node-eth-lookup` module to access the Ethereum network via Infura
- "HOST"
  the DNS / IP address used by the `guardian` to set up the server
- "PORT"
  used by guardian server to set up the port where to listen to incoming API requests
- "REMOTE"
  used by guardian client to set up a manual connection with another Guardian for testing

Important maps used for the global state:

- "genesis_map" is a map of genesis hashes to `StateNodes` with a strong link to anchor the pointers ensuring the StateNodes are not deleted
- "state_forest_map" is a map which maps from a revision hash to the `StateNode` with a weak link
- "contracts" is a map containing weak links to all effective `ContractNodes` indexed by their respective revision hash
  - `ContractNode` contains `ContractData` and a map of hashes to weak links of `StateNodes`
- "shared_revs" is a map that assigns a weak link to `ContractNode` to a hash (shared by contract revision). It is a map for runtime optimization when adding a revision into the `StateNode` map to avoid iterating through all contracts

- “guardian_identities” is a map that assigns Ethereum Account of Guardian to Guardian certificates. This map is populated using the TLS Certificate.
- “user_lookup” is a map that assigns a weak map of hashes which point to the `ContractNodes` (1:n relationship between user and contracts) to the Ethereum public addresses of the users
- “guardian_servitude” is a map of the Guardian’s Ethereum account to the user’s Ethereum account. This map is populated using the Guardian Servitude.

The Guardian needs to keep an internal state of the Aqua-Chains as well as their verification context. For this, `guardian` application builds a tree structure in memory in which the different connected revisions are linked to each other in a data-structure of `StateNodes`. The `StateNodes` represent all revisions that have been requested from the PKC, verified and finally checked if they are part of an Aqua-Contract.

`StateNode:` Represents a single node inside the tree structure which stores all required information on the Guardian’s state. The `StateNode` includes the following data:

- previous node
- leafs of the node
- user accounts for that the StateNode is shared
- if the node itself is the target of a valid and finalized contract entity, the Data Access Agreement is referenced within the node

`StateNodes` are organised in a tree structure to create a StateNode forest. The usage of “weak” and “strong” links enables the correct removal of `StateNodes` during runtime. As seen in the StateNode Diagram, this is achieved by using `Arcs` in combination with `weakMaps`.

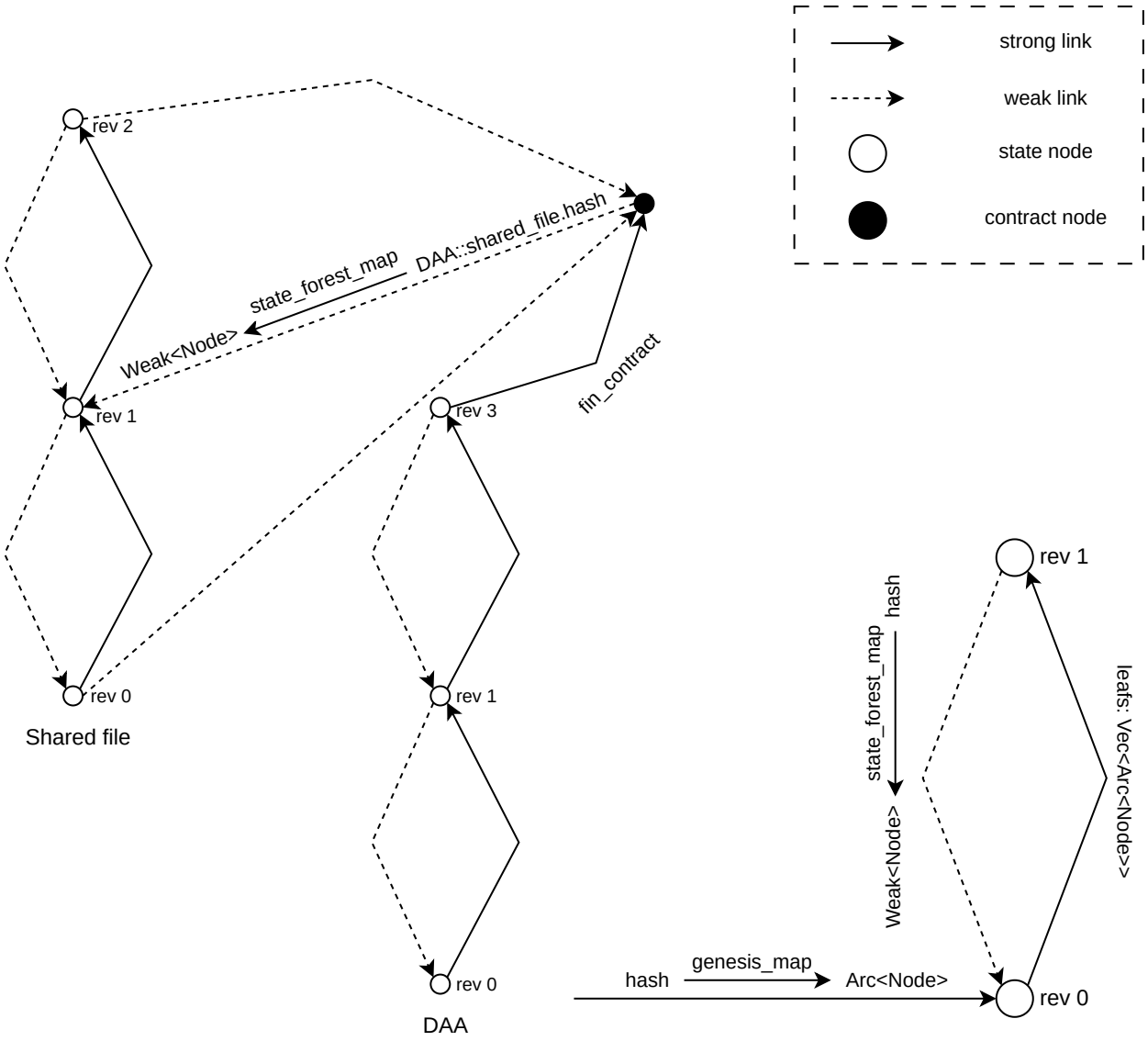


Figure 3: StateNode Diagram

The StateNode forest is accessed using the the following functions:

- `add(Revision)`: this function adds a revision to the `StateNode` forest.

    1. The revision is processed by the verifier module which adds verification data to the `StateNode`.
    2. The revision is processed by the `contract-interpreter` which returns contract data that is stored in the `StateNode`.
        (a) If the `StateNode` represents a valid contract it will get a strong pointer to a `ContractNode` that is spawned with the `ContractNode` data.
        (b) If it represents a valid Data Access Agreement then the hash of the shared revision is added to the `shared_revs`.
        (c) The receiver of the target revision of the DAA is added to the user accounts of the `StateNode` of the shared revision itself.

(d) All child and parent nodes receive the same update of their user accounts for permitting access.

3. It is checked if the added `StateNode` has a previous revision.
   (a) If it has no previous revision, it contains a genesis hash which is stored in the `genesis_map` leading to the creation of a new tree in the `StateNode` forest.
   The entry in the `genesis_map` is a strong pointer to the first `StateNode`.
   (b) If there is a previous revision, the leafs of the previous node is updated by adding a strong link to the currently constructed `StateNode`.
   (c) It is checked if it has become the latest revision, if this is the true the list of latest revisions is updated in `ContractNode` using a weak pointer.
   (d) It is checked if the previous `StateNode` is shared. If this is true the node inherits the list of users for whom the previous node is shared.
4. The `StateNode` is added to the state_forest_map.

- `remove(hash)`: removes a `StateNode` from the StateNode forest.
  1. It is checked if a previous revision exists.
     (a) If this is the case:
        i. The strong pointer of the previous revision is removed.
        ii. It is checked if the previous node has other leafs, if not this previous node has become latest node in the branch and if it is part of a contract the contracts latest hashe needs to be updated.
     (b) If it has no previous node it is the first node (identified by a genesis hash) of a tree and is removed from the `genesis_map`.
  2. If the node had a strong pointer to a `ContractNode` it will be removed.
- `get(hash)`: returns a weak pointer to a `StateNode` defined by `hash`

Functions for permited access used by the Guardian:

1. `accessible_latests(user, hash, owner)`: returns latest revisions which are accessible for a user.
2. `accessible_branch(user, hash, owner)`: returns all revisions before the specified hash to the genesis hash from the `StateNode` forest.
3. `get_rev_accessible(user, hash, owner)`: returns the data of a revision if permitted. Checks if `ContractNode` and if the `StateNode` that is shared by the contract is present.

2.3.2 `verifier`

The `verifier` handles the verification of Aqua-Chains for datastructure integrity, signatures and Merkle Trees.

This component needs to calculate and check all hashes, signatures and witness events (including the Merkle Tree). The full witness event verification is implemented utilizing the eth-lookup module. It should distinguish between different failure modes for the inconsistencies that can arise in Aqua-Chain verification. The different failure modes are important for debugging of not just the Guardian itself but also aligned projects based on the Aqua-Protocol. They additionally inform the user of the different possible manipulations that would render a chain partially or fully untrustworthy.

Interface:

- `fn verify(revision) -> validity`
- `async fn verify_with_witness(revision) -> validity`

2.3.3 `contract-interpreter`

The `contract-interpreter` parses Aqua-Chains' content to extract contract attributes into predefined data structures and verifies the structure of the data against specified conditions to consider it valid.

Contract templates define types of contracts (e.g. Data Access Agreement, Hanshake, Identity Claim) which are known to the Guardian.

MediaWiki's templates are used as Aqua-Contract templates for an improved user-expierence and a strictly defined contract format. Contracts are identified by being instances of a contract template and thus having a template-specific transclusion hashes.

Known contracts are parsed into the corresponding data structure containing the attributes of the contract. Contracts with unknown contract templates are not parsed.

Interface:

- `fn from_revision(Revision) -> Contract`
- `fn identify_revision(Revision) -> state`
- `fn is_contract_effective(iterator over [(contract, state)]) -> effectiveness`

2.3.4 `storage` module

Implements importing and exporting Aqua-Chains from/to a storage provider.

- `pkc-api`

  Implements a storage module by wrapping the import/export APIs of the PKC.

  As the PKC requires authorization for write access, the `pkc-api` will depend on the `siwe-oidc-auth` component below.

The storage module is composed of 6 functions to be used by the guardian:

- `read`

  Takes a revision hash and returns the revision belonging to the revision hash.

- `get_context`

  Takes a revision hash and returns the corresponding context, context is required to store revisions in the PKC (contains Page title, etc.). This API can be deprecated in favor of `get_branch()`

- `store`

  Stores a given revision into the PKC. requires context.

- `list:`

  Returns a vector full of all latest revision hashes in the PKC.

- `get_branch:`

  Takes a revision hash (ideally the latest) and returns a vector of all of its predecessors. It also provides context, returning name-space and title of the article which corresponds to the branch.

- `update_handler:`

  Keeps the Guardian up to date on changes in the PKC.

2.3.5 `eth-lookup module`

Implements functionality to retrieve stored hashes and timestamps from the Ethereum network to provide them to the Guardian application and the `verifier` component. The module returns the block timestamp and the witness_event_verification_hash used to verify an Aqua-Chain witness event.
Interface:

- `async fn lookup(ethereum_chain_id, tx_hash) -> (timestamp, event_hash)`

For implementation of this functionality we use an Ethereum node to lookup chain data. The eth-looku module interacts with an Ethereum node through its JSON-RPC API to retrieve Ethereum transaction data. This provides us with a reliable option to retrieve large amounts of chain-data. Drawback is that we need a self hosted Etheruem node or use a service-provider like infura or alchemy.

2.3.6 `guardian-api`

We have defined a Rust API for the external Guardian API endpoints defined on server and client side (as Guardians form a p2p network). For the client side there is an https method-call and the server has a handler for each endpoint.

2.3.7 `siwe-oidc-auth`

The Guardian needs to prove its Identity to the PKC. This is done with the siwe-oidc Ethereum standard. There is a reference implementation for siwe-oidc available using the siwe crate

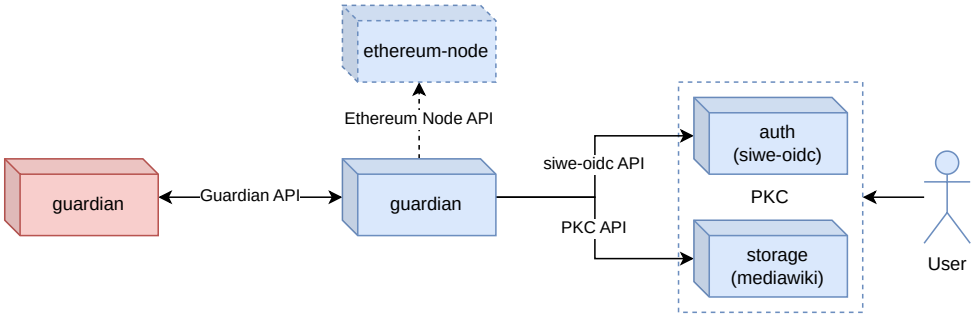
For details on `siwe-oidc` refer to the external API section `siwe-oidc` API.

2.3.8 `guardian-common`

The `guardian-common` component is a Rust crate for module interface definitions, datatypes and encodings for the other components to depend on for adhering to the DRY (Don't Repeat Yourself) principle.

2.4 External Interfaces

For the Guardian to operate it requires connectivity with other services. The services' APIs used by the Guardian are described within this sections.



The diagram shows the used external interfaces. The Guardian communicates with the PKC over the PKC-API which is used to read and write Aqua-Chains from the storage. The PKC API requires authorization which is accomplished by interacting with the `siwe-oidc` server. The Ethereum Node API is used to receive Ethereum transaction data to verify witness events. Guardians communicate with each other by using the Guardian API which is part of the design and implementation work of this project.

2.4.1 PKC API

The PKC REST API is used for importing and exporting Aqua-Chains to and from the PKC.

- `/data_accounting/request_hash/{rev_id}`

  Request the Aqua-Chains verification_hash corresponding to the MediaWiki revision-id.

- `/data_accounting/import`

  Is used to import Aqua-Chains.

- `/data_accounting/get_revision/{verification_hash}`

  Is used to export the Aqua-Chains data corresponding to the revision_hash.

- `/data_accounting/get_branch/{verification_hash}`

  Is used to return the revision_hashes of an Aqua-Chains branch from the provided hash, returning a list of hashes down to the genesis_hash.

- `/data_accounting/get_revision_hashes/{verification_hash}`

  Receive all previous verification_hashes belonging to the branch specified by the revision_hash.

We are using the MediaWiki Rest API to look up recent changes within the MediaWiki container.

2.4.2 Guardian API

The guardian-api implements mTLSv1.3 for transport layer security and uses JSON with REST-API for data exchange. Please find in the appendix an extensive analysis and rational for the mTLS usage. The following endpoints are defined:

1. `list() -> Vec<LatestRevisionHash>`

   Request a list of shared Aqua-Chains from another Guardian. Returns latest Revision hashes:

   (a) Lookup shared Aqua-Chain list from memory for corresponding remote Guardian git

   (b) return list

2. `get_branch(revision_hash) -> Vec<RevisionHash>`

   For an Aqua-Chain requests all Revisions back to the genesis hash:

   (a) check if Guardian is authorized to share revision with `check_user_access(hash)`

   (b) retrieve all revisions from memory corresponding to Aqua-Chain

   (c) return list of revisions

3. `get_revision(revision_hash) -> Revision`

   Requests an Aqua-Chain Revision:

   (a) check if Guardian is authorized to share revision with `check_user_access(hash)`

   (b) retrieve revision from storage module.

   (c) return revision

2.4.3 `siwe-oidc` API

The `siwe-oidc` API is used to talk with the Sign-In-With-Ethereum SIWE Server who acts as an identity provider to provide a session token to the Guardian which is used for logging into the PKC. Open-ID-Connect. We use two endpoints to receive and solve the challenge.

2.5 Control flow

This section explains how we implement the Guardian functionality based on different sequences flows.

2.5.1 Guardian Initialization

The following startup sequence is required before the Guardian is ready to begin to communicate with other Guardians. We build the internal state of the Guardian following this sequence. Guardian startup sequence is described as the follows:

1. Convert seed phrase from environment variable into Ethereum private key (1)
2. Initialize `storage module`(2)
3. Use storage module to get a list of all latest revision hashes (3)
4. Start event-based storage verification through regularly calling the `update_handler()`(4)
5. Get branch for all latest revision hashes to return all revision hashes (5)
   (a) Get each revision from from storage module
   (b) Run through `verifier` (fail->stop) (6a/6b-7b)
   (c) Store verification information in memory (7)
   (d) Run through `contract-interpreter` (8-10)
   (e) Store/update contract information in memory (11)
6. Search for own Identity Claim in memory, if not exists write new self-signed to storage module (12-13)
7. Initialize `guardian-api`(14)
8. Generate API definitions from Guardian configuration and write to storage module (15-16)
9. Write log to PKC(18)
10. Regularly request latest revision hashes from other Guardians via `list()` function (19)

2.5.2 Handling of storage module events

Whenever Aqua-Chains are changed with an Aqua Container we need to verify those new Aqua-Chains and update the internal state of the Guardian.

Trigger Event:
storage-module notifies of new revision hashes or added signatures and witness_events to existing revisions.

Handler:

1. request revision data from storage module (1-2)
2. run through `verifier` (fail->stop) (3a,3b,4b,3.1)
3. store verification information in memory (4)
4. run through `contract-interpreter`(5-7.1)
5. store/update contract information in memory (8)
6. log results (9)

2.5.3 Handling of incoming guardian-api requests from client

This section describes the implemented API endpoints within the server component of the Guardian which are used by the client side.

The following functions are called in sequence from a remote Guardian client to retrieve revisions.

Trigger Event: Guardian API notifies of new revisions.

Handler:

1. Retrieve list of available latest revisions from Guardian via the `list()` function. (1)
2. Retrieve branch via `get_branch(revision_hash)` for each new latest revision hash. (2)
3. Retrieve all new revisions via the `get_revision(revision_hash)` function: (3)
   (a) verify incoming revision (4a, 4b, 5b, 4,1)
   (b) store the retrieved revision in the PKC via the storage module (5-6)

3 Design Decisions

In this section reasoning for design decisions and alternatives are described.

3.1 Programming Language

Decision: Rust
Alternatives: JavaScript/TypeScript, C/C++, Go, Python

Factors:

- Language familiarity

  At least some members need to be very familiar with the language in order to plan the Architecture accordingly and to set realistic goals.

  A greater share of the group members will have previously used C/C++ than Rust, as such it might seem a risky choice. Rust however trumped C/C++ in the maximum depth of knowledge in the group and provided greater benefits in the following factors.

- Security

  The language needs to enable a security focused design so that we can fulfill the requirements set in the Specification Document.

  The benefits of memory safety, static typing and value semantics offered by Rust made it shine in this consideration. The other candidates lacked true memory safety and value semantics (no null pointers) which enforces better design decisions and puts less stress on the developer to consider optional values everywhere. Interpreted languages such as JavaScript/TypeScript and Python additionally lack real static typing which enforces more defined interface design.

  JavaScript/TypeScript are known to frequently have supply chain attacks with their package system. Go and Rust are not exempt from this problem, however it is currently less frequent and easier to analyze dependencies. C/C++ sidesteps this issue by not having well established, generally accepted means of package management which is however a downside in itself.

- Libraries

  The Guardian is required to interact with many external APIs and standards. Being able to rely on existing implementations is a must for meeting requirements and deadlines. The correctness and speed of existing implementations wouldn't be matched in this project and would be out of scope.

  Go, JS/TS and Rust offer many libraries for working with cryptographic signatures and hashes, such as Ethereum Accounts and SHA3.

- Efficiency

  Though not a very important factor, still quite desirable; Efficiency is achieved more easily with compiled languages such as C/C++, Go and Rust.

Conclusion:
Even though Rust is far behind other languages when it comes to ease of use it is competitive or superior in all other respects. As we are building an application with security at its core, Rust brings fundamental benefits that very few languages can compete with. It is largely for this reason that we chose Rust.

3.2 Cryptography Standards

Decision: SHA3-512 Hashes, secp256k1, Ethereum Accounts, Ethereum Blockchain
Alternatives: SHA-256/-384/-512, SHA3-256/-384, other hashes, GPG, Bitcoin

Factors:

- Alignment with Aqua

  As the Guardian should integrate into the existing ecosystem provided by the Aqua project, little choice was to be had. Still, the design decisions process used for the Aqua project's choices will be listed.

- Cryptographic Timestamping

  Writing data onto a Blockchain is generally expensive and must be kept low in order to keep the barrier of entry low as well.

  With both Ethereum and Bitcoin being considered as Blockchains, Ethereum won out because of generally lower costs and faster blocktime. Other Blockchains were not considered due to the lower transaction-security.

- Industry Adoption

  The problem of tools aiming to solve somewhat aligned issues has been adoption and usability.

  With the goal of increasing the odds of success for the Aqua project, Ethereum has been chosen, as it has one of the largest usercounts for systems with cryptographic Accounts. GPG is the de-facto standard for signing which has however not gained a big userbase despite being integrated into many everyday applications like E-Mail.

  The Ethereum ecosystem provides important, well maintained tooling for private key management and signing with secure and user-friendly wallets. It also has increased momentum and ongoing innovations like Ethereum multi-signature wallets.

- Security

  To align with the requirement of security, used hash functions need to meet all requirements of cryptographic hashes.

  The SHA3 hashes were preferred over SHA2 and other older cryptographic hash standards because of the recommendations from NIST. SHA3-512 was chosen over smaller bit sizes to provide minimize the risk of a collisions.

  We are aware of post-quantum security risks for (mainly) secp256k1 public key encryption, which would require key rotations and TTL counters for signatures to reduce risks, this is however outside the scope of this project.

- Cohesion

  The cryptographic standards should be similar and interoperable so that users do not have to buy into multiple competing standards.

  With the Ethereum Blockchain already chosen, Ethereum Accounts' cohesion was prioritized over GPG's wider integration into everyday applications and possibly adoption.

  The Ethereum and Bitcoin Accounts use secp256k1. The shared Keccak family of both SHA3 and secp256k1 also shows good cohesion.

Conclusion:
Dominant factor is alignment with the Aqua reference implementation. Furthermore are we using well established and NIST specified encryption. The encryption ciphers are well documented and tested. Various implementations and tooling is available for implementing those standards into the Guardian. We are using well established wallets to manage Ethereum accounts and rely on the Ethereum community to provide secure tooling.

3.3 Data Storage and Frontend

Context: For the Guardian to be able to manage data, there has to be a place where the data is stored. This section describes our choice on this matter and what we factors we considerd.
Decision: MediaWiki with mediawiki-extension-aqua
Alternatives: Nextcloud

Factors:

- Familiarity

  Members of the group should be familiar with the software in order to plan the architecture accordingly and to set realistic goals.

- Software Maintenance, Support and Ecosystem

  MediaWiki is well maintained and funded by the MediaWiki Foundation. As long as Wikipedia remains a regularly used website with sufficient regular funding we can expect MediaWiki to be maintained. Furthermore MediaWiki has a broad ecosystem of extensions and developers which are familiar with it.

  Nextcloud has a similar or even better support for their Software as they operate as an enterprise software provider for many customers. They provide ongoing development and support for their software.

- Open Source and Accessibility

  Both MediaWiki and NextClound are open source.

  Nextcloud is a well established open source software with very well written documentation and available ressources like extensions for various functionalities.

- Extension for Aqua-Support

Mediawiki-extension-aqua already exist and supports the features we need (exporting and importing Aqua-Chains). Nextcloud on the other hand lacks support and would require a similar extension. The development of such an extension for Nextcloud is out of the scope of this project.
Furthermore mediawiki's editor is already extended for adding cryptographic signatures and timestamp information.

- Alignment with Aqua-Chain structure

  MediaWiki is a wiki with a large toolset to manage interlinked pages. MediaWiki revisions align to Aqua-Chain revisions. The similarity between Aqua-Chains and MediaWiki pages makes the integration comparably easy. Nextcloud however lacks persistent revisions, making it significantly harder to implement a seamless integration with our currently planned architecture.

- Data Storage and Management

  Nextcloud supports storing all file types very efficiently. Nextcloud has a capable permission system, whereas MediaWiki is designed to be public. MediaWiki supports common file types typically embedded into wiki pages. At least for a prototype this should be more than enough. MediaWiki proved to be capable of large-scale collaborative knowledge management.

- User Interface

  MediaWiki has an interface for text files (specifically wiki pages). MediaWiki supports many file types for previewing and including. This is very convenient as a large part of the files the Aqua protocol will deal with are supported.

  Nextcloud has many user tools for managing and editing stored data, but the coverage of our use-case is no better than with mediawiki.

- Offline Capability

  Both work offline.

Conclusion:
While Nextcloud may be better for dealing with files in general MediaWiki is better suited for the uses-cases we care about most.

Revisions especially are very important for our purposes. Additionally, the fact that MediaWiki already has preexisting infrastructure for the Aqua protocol is extremely convenient. For these reasons we decided to use MediaWiki for our prototype but hope to keep Nextcloud as a future possibility.

3.4 Guardian API Transport Layer

See appendix for mTLS which is used for transport layer security and access control for the Guardian API.

Appendix mTLS

Guardian Handshake - Certificate Exchange

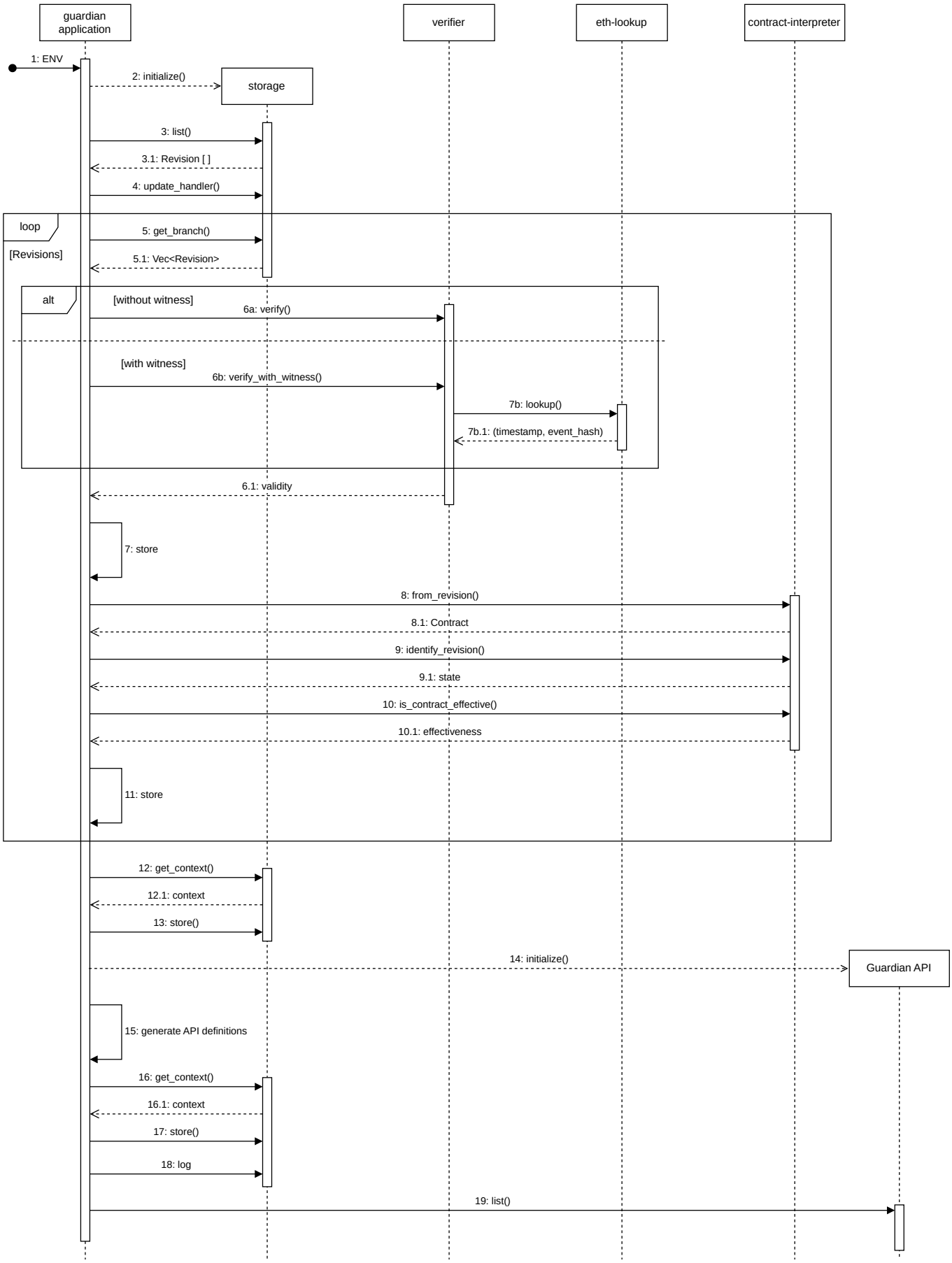


Figure 4: Control Flow Diagram for Initialization

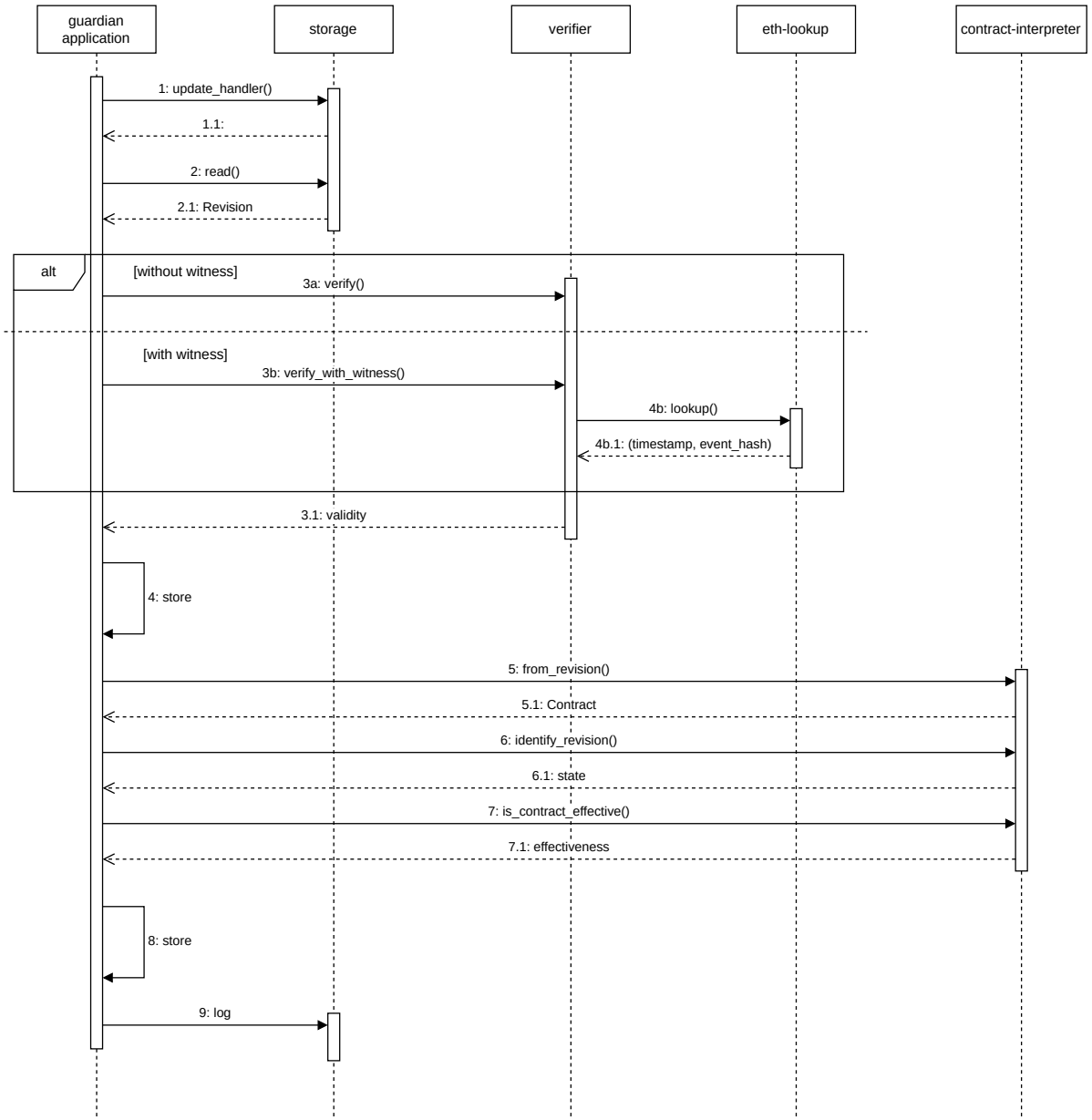


Figure 5: Control Flow Diagram for handling `storage` event

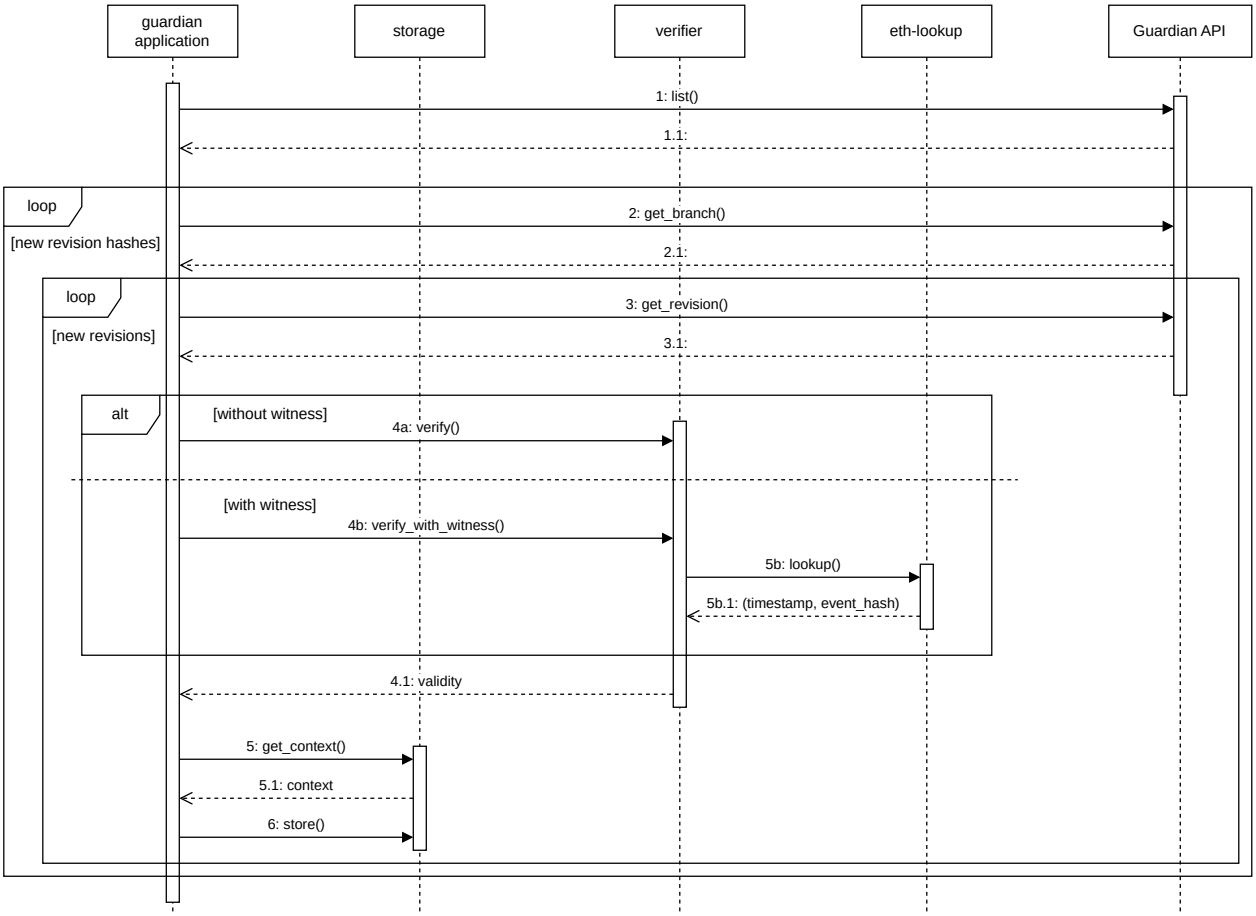


Figure 6: Control Flow Diagram for handling incoming `guardian-API` requests from client