Harpokrat Technical documentation

{EPITECH.}

LE FUTUR DE L'INFORMATIQUE
LE MEILLEUR DE L'INNOVATION

EPITECH INNOVATIVE PROJECTS

Summary of the document

This document is the technical documentation of Harpokrat. It is written for any potential contributor to the Harpokrat development process and requires for the most part a certain technical knowledge.

The first part to acknowledge is the secret sharing which is required to understand the architecture of the API resources used in the different parts of the project.
We also present the different clients of Harpokrat. There are four different clients with the web one being the most important, with all the functionalities of harpokrat implemented. The mobile client is in Flutter and will therefore work on both IOs and Android. The browser extension is compatible with chromium based browsers as well as Firefox (meaning we cover most of the existing browsers). Last but not least the CLI client allows the user to use Harpokrat and recover or store passwords directly from a command line interface.
Next, we talk about the API that handles the storage part of the project, in other words, it stores all the encrypted secrets produced by the various clients using the Harpokrat Cryptographic Library (HCL). The API is also responsible for the access control management but, as our zero-trust design induces, we do not rely on this for our core security.
The explanation of the internal libraries will follow like the Harpokrat Cryptographic Library (HCL) which is the core library, responsible for all the cryptography related mechanisms and algorithms of the project, thus making it the central element of Harpokrat technology. We also talk about the Angular and Javascript libraries which are used by both the web client and the browser extension.

As we explained this documentation mainly targets the technical team of Harpokrat and whoever wants to learn more about the core working principles of Harpokrat.

About this document

Document information

Title	Harpokrat Technical documentation
In charge	Fantin Bibas <fantin.bibas@epitech.eu>, Nicolas Lemaire <nicolas.lemaire@epitech.eu>, Tanguy Gérôme <tanguy.gerome@epitech.eu>, Louis Mallez <louis.mallez@epitech.eu>, Faudil Puttilli <faudil.puttilli@epitech.eu>,Antoine Stempfer <antoine.stempfer@epitech.eu>
Last update	2020/12/113
Version	1.0
Keywords	Harpokrat, technical, documentation, usage, manual, guide, development, code, API

Revision table

Date	Version	Section(s)	Details
2020/08/01	0.1	All	Creation of the base of the document
2020/08/16	1.0	All	Add the content of the documentation
2020/12/13	2.0	All	Add and update content based on feedback from previous document delivery.

Group organization

Each member of the group has taken part in writing this documentation, with overall everyone documenting the part they developed :
Antoine: Javascript Library, Angular Library, Web Client
Fantin: Backend API, Secret Sharing, HCL
Faudil: Mobile App
Louis: Backend API, Python Library, Deployment
Nicolas: HCL, Python CLI
Tanguy: Web Extension

Some documentation was needed that did concern everyone in the group, and for those we just all participated on a voluntary basis.

Table of contents

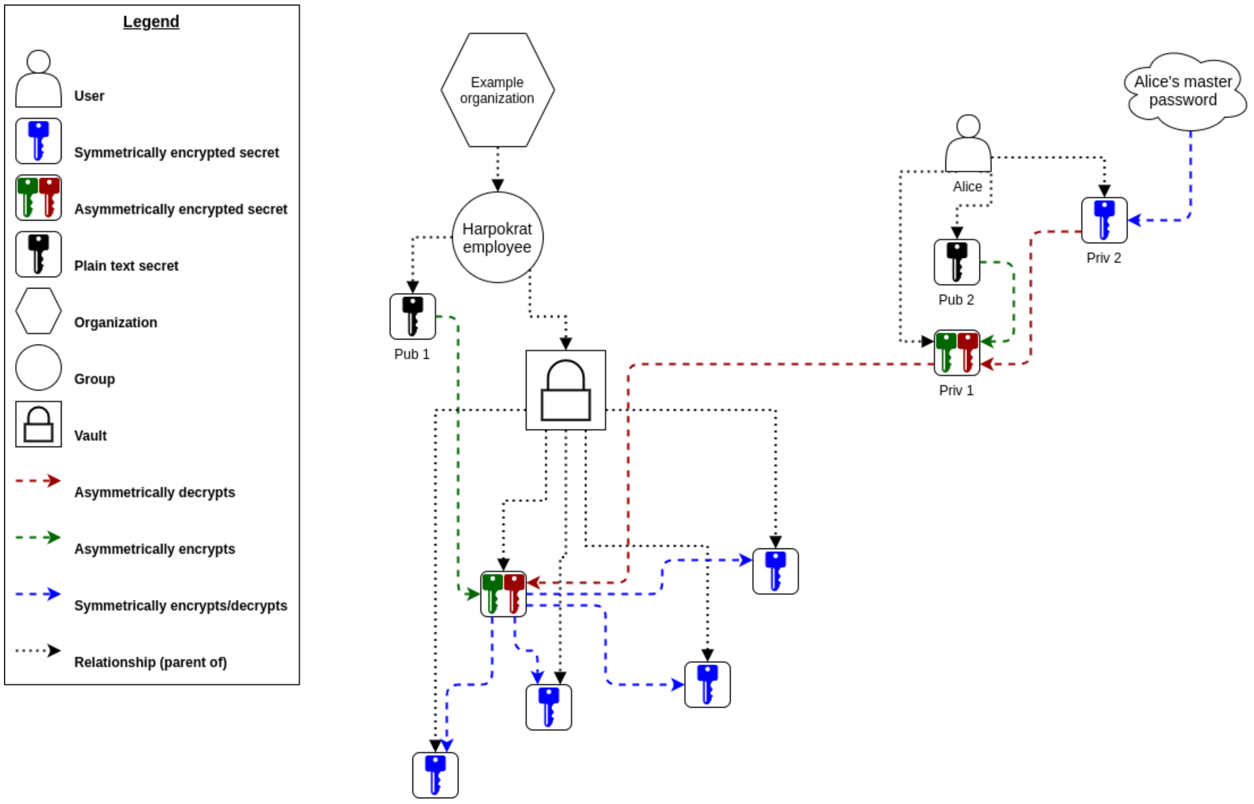
Secret sharing

The secret sharing is a big part of what will make Harpokrat great since it is a vital tool for the use of the solution by companies of all sizes.
This first part of the documentation is purely theoretical but is needed to be well understood to understand any part of the project.

We will explain how we securely share secrets between the employees of a company while keeping the possibility to change the access rights of a user and have fine tuning of which secret which user can have access to.

A simple example

In the following illustration of a simple example of password sharing relations you can see how the relations between the different API resources works.



Simple example of password sharing relations

Let us now try to understand this illustration.

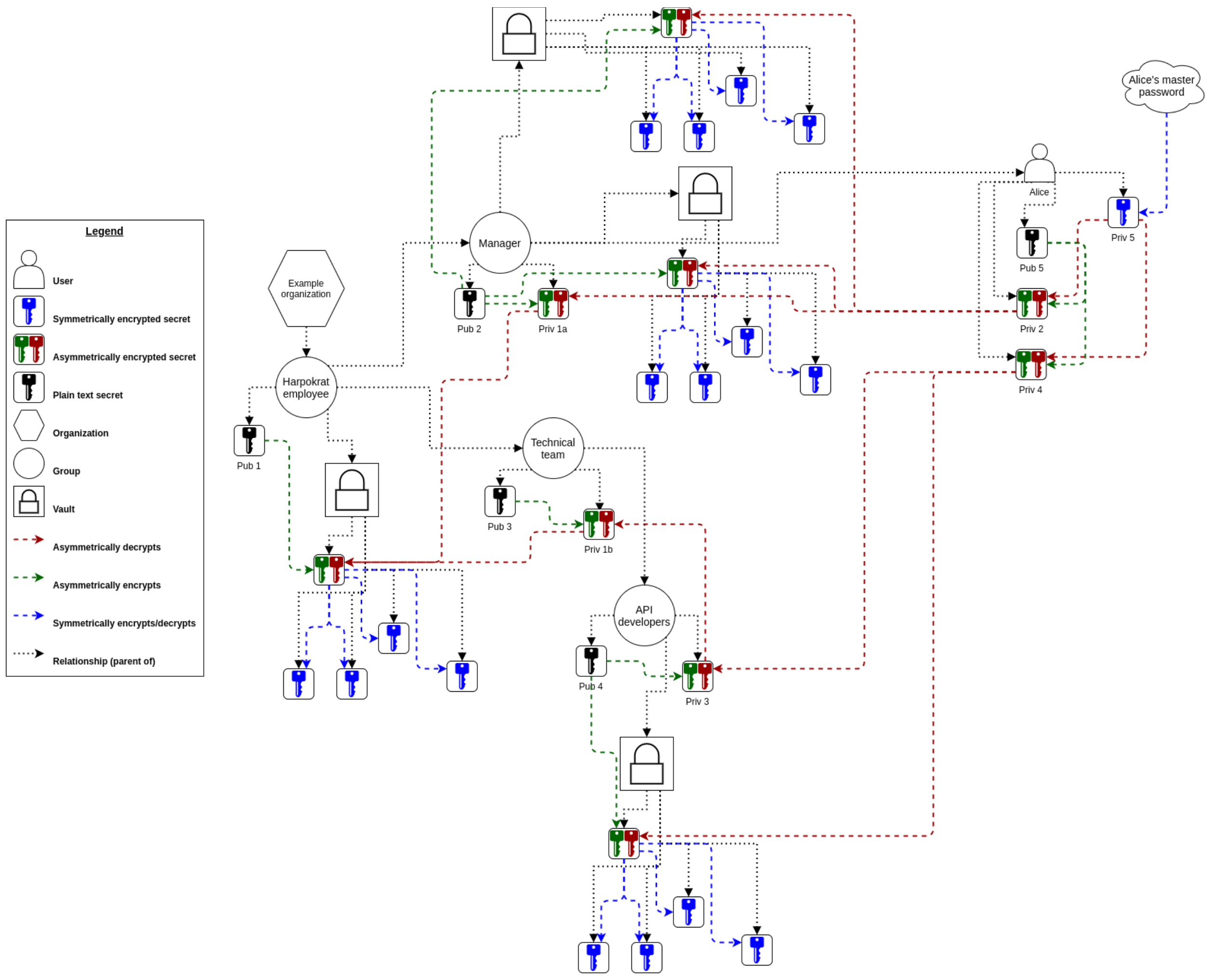
The first thing to understand is that all the keys of any type are looking the same for the server: a big blob of data that cannot be understood. This is why Harpokrat is so secure: even if we got compromised, it would not be a problem since we cannot have access to any key content. This is also why our solution can be used by our clients without the need for them to trust us.

Another important thing in this illustration is to well understand the legend: the different key representations does not indicate what is the content of the key but instead how the key was encrypted. The keys can be stored in 3 different formats:

- ➔ Symmetrically encrypted: using a symmetric algorithm, meaning using the same key for encryption and decryption
- ➔ Asymmetrically encrypted: using an asymmetric algorithm, meaning using a different key for encryption and decryption. The key used for encryption is called a "public key" and the key used for decryption is called a "private key". This is because encryption data is something that could be performed by anyone, it is public, but decrypting the data should only be performed by the owner of the key, it is private
- ➔ Plain text: the key is not encrypted. This is only used for asymmetric public keys since these keys are really public meaning anyone should be able to access it. There is absolutely no confidentiality security concern for this kind of key

Let us now talk about how the relations between the entities work in this password sharing system.
We will begin by the Vault. The Vault is the resource that will own all the secrets that a company possesses. It contains at least one secret which is the master password of the vault. This secret is the key used to symmetrically encrypt every other password present in this vault. The master password of the vault is asymmetrically encrypted using a public key owned by the owner of the vault (the owner of the vault can be a group or a user). This asymmetric encryption allows certain members of the group (defined by access control rules defined in the organization and group settings) to update the master key (and therefore update all the passwords of a vault) for example when an user is removed for a group so he cannot cheat the system by using the old password (even if the access control already prevent the user from getting the encrypted data, this is just an additional measure in the unlikely event of the secure storage of Harpokrat getting compromised).
In the same way, every user possesses a private and a public key. The public key of the user allows the managers of a group to share the private key of this group with the user. The private key of the user is encrypted with its master password.
And this is basically it for this simple example. Let us now discover a more advanced scenario.

Advanced hierarchy



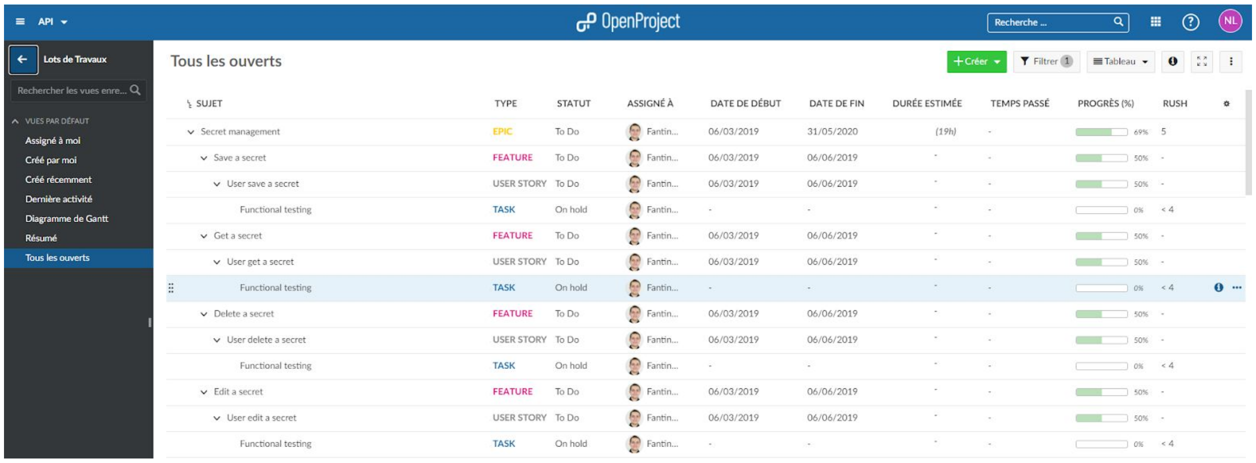
An advanced example with use of group inheritance

In this complex illustration we can discover the group inheritance, allowing for more advanced hierarchy management. The example is very close to the previous one with the exception that it illustrates a more realistic scenario: group inheritance. This feature allows a group to be the owner of another group. The owned group possesses a private key that allows to decrypt the private key of the parent and thus accessing all its vault master keys and consequently all of the stored secrets in those vaults. This is obviously recursive and a user having the private key of a group can escalate from one private key to the other to get access to the full hierarchy of groups.
In the illustration you can also see that a user can have multiple private keys corresponding to different groups and can be part not only of the deepest group of a hierarchy but also of a parent and therefore without the access of its children groups.

This architecture and hierarchy system allows a company to easily implement business related password sharing applications by creating groups with a real life meaning. This will be very simple to use even for a non technical manager in a company.

Organisation and communication

In order to organize all HPK's tasks, we use a great tool called OpenProject. It is accessible at https://management.hpk.company/. It allows us to keep an eye and track easily the progress of each task.



Tous les ouverts

SUJET	TYPE	STATUT	ASSIGNÉ À	DATE DE DÉBUT	DATE DE FIN	DURÉE ESTIMÉE	TEMPS PASSÉ	PROGRÈS (%)	RUSH
Secret management	EPIC	To Do	Fantin...	06/03/2019	31/05/2020	(19h)	-	69%	5
Save a secret	FEATURE	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
User save a secret	USER STORY	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
Functional testing	TASK	On hold	Fantin...	-	-	-	-	0%	< 4
Get a secret	FEATURE	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
User get a secret	USER STORY	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
Functional testing	TASK	On hold	Fantin...	-	-	-	-	0%	< 4
Delete a secret	FEATURE	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
User delete a secret	USER STORY	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
Functional testing	TASK	On hold	Fantin...	-	-	-	-	0%	< 4
Edit a secret	FEATURE	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
User edit a secret	USER STORY	To Do	Fantin...	06/03/2019	06/06/2019	-	-	50%	-
Functional testing	TASK	On hold	Fantin...	-	-	-	-	0%	< 4

Moreover, we use Discord (https://discord.com/) to organize all kinds of meetings between all the members of Harpokrat. It is a voice server where you can easily share information such as your screen when you need it.

Sources & Version Control System

To manage code for the whole project, we use Git repositories.
All our repositories are hosted in a GitHub organization (https://github.com/harpokrat-company), as this integrates easily with CI/CD tools (see Deployment below). If you need a new repository created, ask Fantin or Louis, as they are the authorized managers of the organization.

Norms

To be able to have clean integration with CI/CD tools later on, we need git repositories to be maintained in a clean state. For this, our preferred workflow is the very common "Git Flow".
To get information on it, follow this link:
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

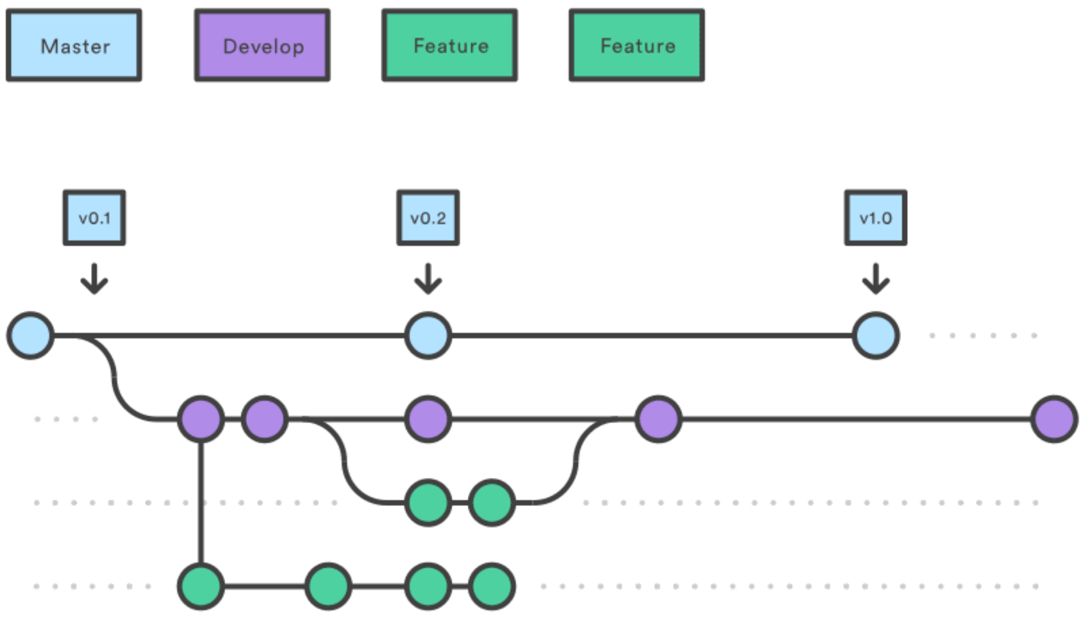
As for actual code writing norms, we have not yet felt the need to define actual norms, but we usually try to follow what is considered best practice per the language used in any part of our codebase.

Deployment

All parts of the project respect the principle of continuous integration. Indeed, we used the CI tool provided by Github, called Github Actions. It automatically runs a test suite and deploys on our servers.

A quick documentation of Github Actions can be found on this link.

In order to properly set up a continuous integration, it is very important to use a uniform git branching model. Thus it is easier to automate the deployment by deploying specific versions using tags (according to our set Git Flow Workflow).



Web client

The web client is made using in typescript using the Angular framework which allows to make Single Page Applications (SPA). We also use a customized version of Bootstrap to style the website.
The web client mostly depends on the components available inside the Angular Library so the code for the web client is rather small.

There is a component for each screen that uses the different components from the library.
The only part of the web client that doesn't use components from the library is the home page that presents various information about the company.

Mobile client

The mobile client is developed for IOS and Android. The framework used is Flutter, a cross-platform framework created by Google to create native applications. The main advantage of using this technology is that the application is rendered into a standard canvas before being displayed and doesn't use any of the system's GUI components. This ensures that the application will look the same no matter the OS and spare a lot of time for testing on both platforms. The android jetpack API has been used as a base layer for the android app to be compatible with the android version from 4.0 to 11.

Code norm

This project follows the standard Dart coding style:
https://dart.dev/guides/language/effective-dart/style

The architecture

The architecture follows the guideline of MVC (model view controller). Because there is not a lot of types of data to Handle (The User and its data) there is only one controller (Session).
The model data structure comes from the HCL (Harpokrat Cryptographic Library). But because it is written in C++, we use a wrapper (HCL flutter wrapper) written in Dart which is just plugged into the application. This wrapper is compatible with IOS and Android but only has been tested for Android. This architecture is simple and straightforward as it can be: one class per view and one class per model.
A library is used to autocomplete other apps on the phone. But because there is none which is cross compatible. It only has been coded for Android in Kotlin using the Android's autofill API.
The autocompletion part has been coded in Java and all of the code is in a single class named HarpokratAutofillService which inherits from the android system class AutofillService. The Java code communicates to the Flutter code through a method channel invocator.

Dependencies and framework used

Rest_api: A wrapper around the http library to better handle requests and manage the relationships and resources provided by the API.
Url_launcher: Deprecated in the last versions of the android sdk and will soon be replaced but useful to launch the web browser in the app.
SafetyNet: (Only for Android) because we need a valid captcha to create an account we decided to use the SafetyNet captcha API provided by Google Play Services.

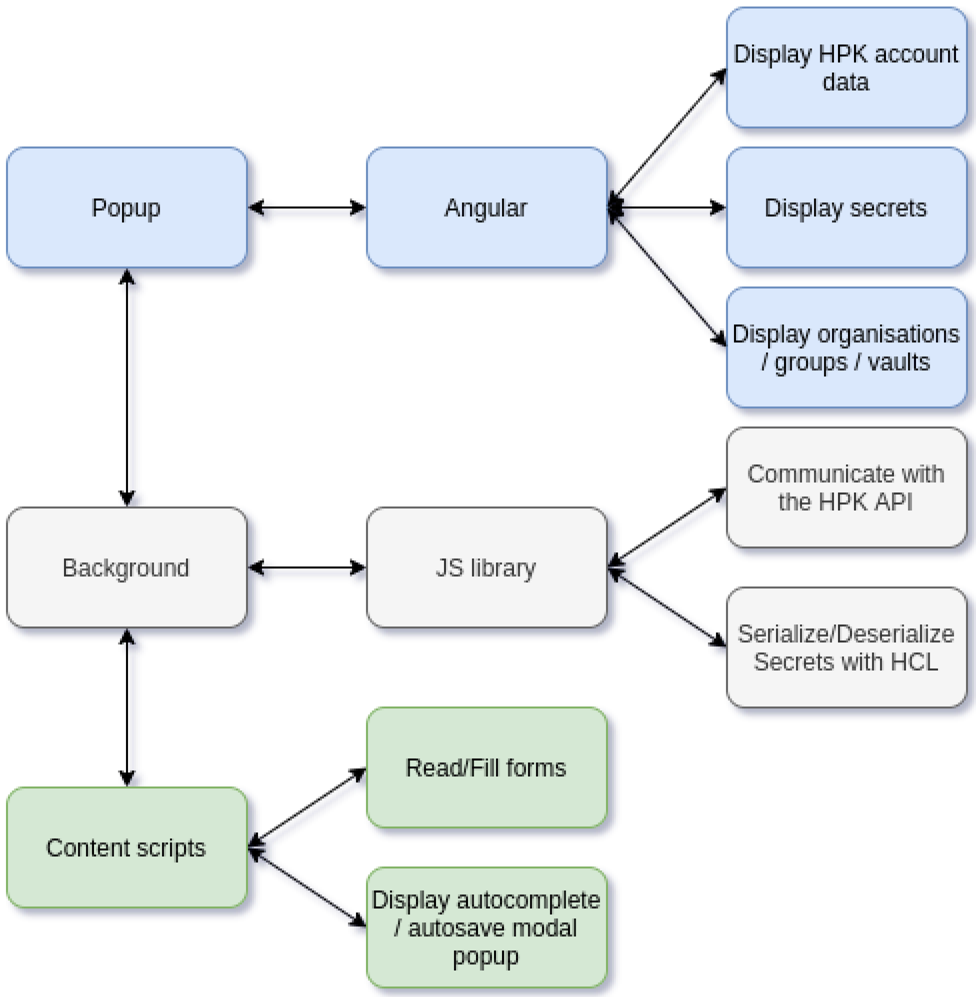
Browser extension

The browser extension is written using the Web Extension framework. This is a somewhat universal way of writing browser extensions, but Google Chrome does not entirely respect it. To address this, we wrote a simple unification tool that converts a “standard” Web Extension to be usable in a Chrome environment (see later “Unification tool” section).

Extension code organisation

Web Extensions are divided in 3 main parts:

- The Popup in what you see when you click the extension’s icon in the top right corner of your browser ;
- The Content scripts are javascript pieces that are injected into the web pages you are browsing ;
- The background script is the extension code that always runs while the browser is running and the extension is installed.


Popup
Angular
Display HPK account data
Display secrets
Display organisations / groups / vaults
Background
JS library
Communicate with the HPK API
Serialize/Deserialize Secrets with HCL
Content scripts
Read/Fill forms
Display autocomplete / autosave modal popup

Popup

The Popup is written using Angular, and a graphics only Angular library. This is used to show data to the user in a dashboard. The popup gets the data it shows by communicating with the background script with the `sendWebExtMessage(message_type, params)` wrapper. This async function will resolve into the background script's answer.

Content scripts

The content scripts can communicate with the background through the async `send_webext_message(message_type, params)` (will resolve into the background's response) and `add_message_listener(message_type, handler)` wrappers.

In `content_scripts/find_fields.js`, you can find all the code related to finding login fields and forms, and using them to both catch new accounts to register to an HPK vault ; or complete those fields with already recorded credentials. To add a new "field finding method", add a new

fonction to the array in the `find_fields()` wrapper along with the web domains on which it is effective. This new method will have to find the fields in the web page's DOM through normal javascript means (`document.querySelectorAll("form")` for example) ; and return an object like the following : `{form: ... , user: ... , pass: ... }`
Modal popups are built in `content_scripts/modals.js`: in there we build the elements of the popups, and then add them to a container div, with a big z-index to "pop" above page contents.
`content_scripts/new_account.js` looks for known log in forms, and sets up events to catch new accounts to be saved
`content_scripts/confirm_fill.js` looks for known log in forms, and checks if an account for it is recorded in the HPK vault. If so, it asks if the user wants to auto-complete it.

Background

The background is responsible for communicating with the API, and (de)serializing the secrets, through the API JS library made by Antoine. It then sends needed information to the content scripts and popup, through the `send_webext_message(message_type, params)` (will resolve into the handler's response) and `add_message_listener(message_type, handler)` wrappers.

The background has been converted to typescript, in order to help with maintainability. It is transpiled back into javascript before being fed to the browsers, using WebPack.
Tests are run when the extension is installed and reloaded (in debug mode). Tests can be written by adding an async test function in the `background/tests.js` file. If adding tests, you might want to initialise some data/storage, which you should do in the `background/background.js` file. There is no real way of making much more complicated testing suites in the WebExtension environment (for anything else than background scripts), and so quite a big part of the testing simply has to be done by hand.

Wrappers around the extension storage (storing data on the browser, not the API) can be found in `background/storage.js`.
We wrapped all the code used to communicate with the API in a Web Worker (`background/worker.ts`), to enable us to (de)serialize without blocking the whole browser execution (this is a very long, resource intensive process, and would be noticeable). Communication with this worker is then wrapped in `background/api.ts`, exposing async functions (like for example `async function login(email: string, password: string)` ; resolving to the user's token if login was successful) ; and also messaging events (like `hpk_login`) to be used in `send_webext_message()`

Unification tool

The unification tool is a simple script that will go through the extension's files and adapt it to run on Chrome.

To use it simply run `./generate.py [path/to/web/ext] chrome`
This will create a folder named “chrome” with, in it, the extension you can now run on chrome.

Building, packaging and uploading

To build the extension, just run `npm install && npm build`
You can then load this as a temporary extension in Firefox : about::debugging -> This Firefox -> Load Temporary Add-on ; or on chrome More tools -> Extensions -> Load unpacked.

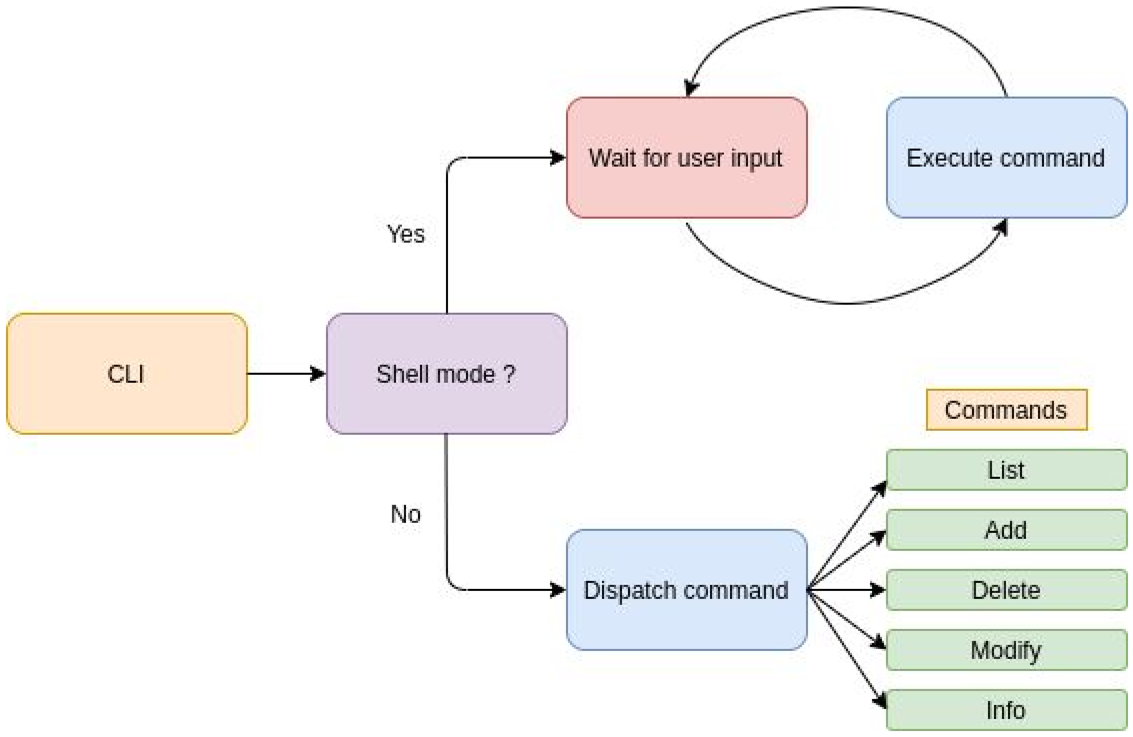
Packaging the extension is easy : just create a zip archive out of it after having build it.
You likely want to keep the sources in the zip (Mozilla’s Extension Store asks for them), but you can and should remove the `node_modules` folder to make it more lightweight.

To upload for Firefox, just go to the “Developper’s Hub” (https://addons.mozilla.org/en-US/developers/), select Harpokrat, and Upload New Version.
Think of correctly incrementing the version number (as well as in the `manifest.json`).

To upload for Chrome, you need to pay a one time personal license, and so we shall keep doing this ourselves (ask Fantin to do it as he has the license).

Command Line Interface

The command line interface is a simple python script that can be used as an interface above the python library that is responsible for calling the API.



An illustration of the general functioning of the command line interface.

With the help of argparse, the argument list is extracted and then parsed in order to be interpreted. The shell mode does not really change the program flow except a loop.
All existing commands inherit a class called command. It is implemented as follows:

```
class Command:
    def __init__(self, label):
        self.label = label

    def get_label(self):
        return self.label

    def print_usage(self):
        print("This command has no usage for the moment")

    def debug(self):
        print("Calling " + self.label)

    def run(self, harpokrat_api, args):
        print("Running " + self.label)
```

Indeed, all the commands have a label that allows us to differentiate it from its siblings and also to match it. For example, the List command is a class that inherits from the Command class and has the label “list”. The method called run should contain all the things that the command should do when it is called.

As previously implied, shell mode is just a simple loop waiting for user input and calling the associated action.

The writing standard used throughout the project is described by the well known PEP n°8. It can be found at https://www.python.org/dev/peps/pep-0008/.

Python packages

PyInquirer

This package is really useful to the CLI. Indeed, it helps to create easily a beautiful command line interface by providing a complete functional set of functions that deals with user input. In other words, PyInquirer ease the process of providing error feedback, asking questions, parsing input and validating answers.
Link to their documentation: https://github.com/CITGuru/PyInquirer

PyFiglet

It is a package that only helps translate ASCII text to an amazingly beautiful ASCII art front.
That’s it.
Link to their documentation: https://github.com/pwaller/pyfiglet

Backend API

The backend API is a major part of the project as it is the only backend element of our design, it will manage the link to the database and allow the data to be accessible everywhere.
Since we are using a no trust design the bookseller will only receive encrypted data and therefore will not be able to know the real values of the passwords.
The API is made in PHP with the Symfony framework (currently in version 5.2 but it will evolve with the framework), we use the file architecture promoted by sensio lab (editors of the framework) and called Symfony Flex, so it is important to know it well in order to work efficiently on the project and understand how it works : https://symfony.com/doc/current/setup.html

Json Api

The API is based on the JsonAPI standard (https://jsonapi.org/), so it is important to get acquainted with it in order to be able to work on the API. In order to be able to use it from a developer point of view with symfony we use a community package : https://github.com/paknahad/jsonapi-bundle. This one includes Hydrator and Documents which allow to convert data in jsonapi format.

Each resource will have its own Controller which will allow to answer to the requests. Its Document to convert an entity in JsonAPI format, one or more Hydrator which will allow to convert the bodies of the answers in modification to be carried out on an entity. In order to facilitate this many abstract classes have been created and implement most of the generic functionality as well as utility functions.

Several resources and many API routes exist and are usable by different clients. These routes all respect the json api and can be found in a document on this link: https://documenter.getpostman.com/view/6844356/T1LFoWF8.

Authentication

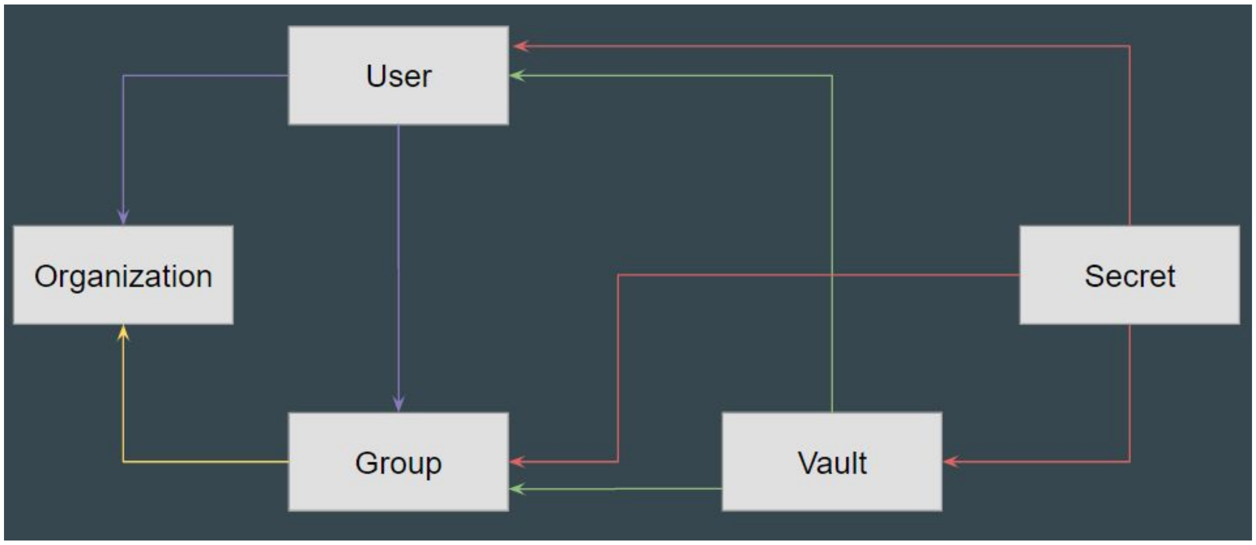
To allow users to authenticate and access their words face-to-face, we use a bearer token system in the API: https://oauth.net/2/bearer-tokens/#. The user will ask with his login and password this one via a POST request on the json-web-token route which will return this token which will be valid for a certain period of time.

Organization Resources

In order to be able to implement password sharing between users several resources have been implemented in the backend in addition to the secrets and users that existed before:

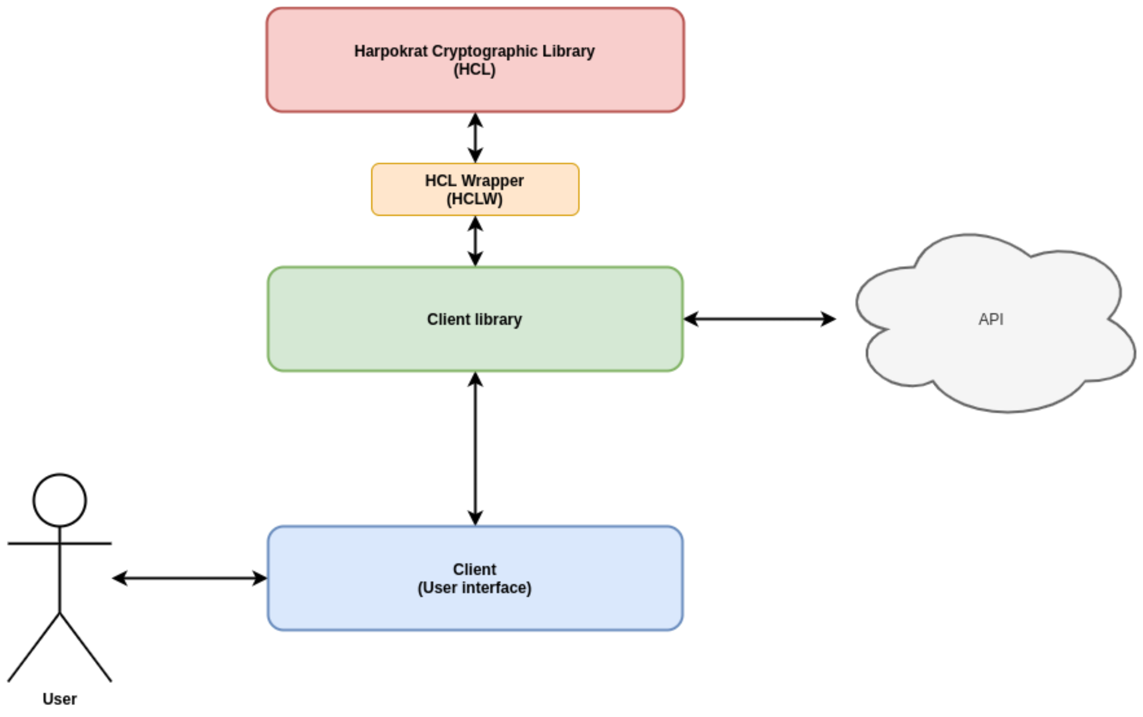
- Organizations: which contain a group of users...
- Groups: which are one level below an organization, these can contain other groups and contain passwords and vaults.
- Vaults: they represent a group of passwords encrypted in the same way

The links of these organizations are a bit complex, you can see them in the diagram below. An Organization contains users and groups; groups contain users that are part of their parent organization. Users and Groups can contain secrets and vaults, a vault also contains secrets.



Harpokrat Cryptographic Library

The Harpokrat Cryptographic Library (HCL) is the main library used in all the frontend clients to handle all the cryptographic functions of harpokrat. This includes mainly password encryption and decryption but also authentication to the API, password sharing, signing of messages and objects and verification of signatures.
Because this library needs to be used by multiple frontend clients in different languages and used in different environments, the library is developed in C++ and compiled in different ways. A wrapper is also developed for each frontend client to help bind the core API functions to a more language oriented API and also handle all the specificities of the wrapping (allocated memory handling, object instantiation and destruction, etc...)



An illustration of the interaction between the different application tiers of Harpokrat.

The technical documentation of the HCL will be splitted between the core and the different wrappers.

Core

The HCL core is the core component of all the frontend clients, handling everything related to cryptography, the core of the business side of Harpokrat.

Binding and export

Because the library is used by different languages, it must be as simple as possible in its exported API to be compatible with different types of programming. For example, the library is in C++ and makes use of classes; however, the exported functions are not object oriented and are plain functions that take an object pointer as an argument. Those functions are called ‘linkage functions” and are represented by the topmost double-arrow in the illustration of interactions between the application tiers of Harpokrat.

As an example, let us take the hypothetical `HCL::Secret` class. The stripped (and very simplified) declaration of the class looks like this:

```
namespace HCL {
class Secret {
 public:
  Secret(const std::string &key, const std::string &content);
  const std::string &GetLogin() const;
  const std::string &GetPassword() const;
}
}
```

However, we don't directly expose the class as the API but instead some linkage functions that will help the wrappers use the library in a more flexible way. Here is what would those linkage functions look like for the extract of `HCL::Secret` class:

```
HCL::Secret *GetSecretFromContent(const char *key, const char *raw_content)
{
    return new HCL::Secret(key, raw_content);
}
const char *GetLoginFromSecret(HCL::Secret *secret) {
    return secret->GetLogin().c_str();
}
const char *GetPasswordFromSecret(HCL::Secret *secret) {
    return secret->GetPassword().c_str();
}
void DeleteSecret(HCL::Secret *secret) {
    delete secret;
}
```

As you can see the class pointer (`HCL::Secret *`) is a simple value that is passed by to the linkage functions. The explanation of how these linkage functions will be used by the wrappers will be detailed in each wrapper section.

We also implemented some utility functions to handle basic objects correctly like `std::string` for example.

Secrets

The `HCL::ASecret` class is very important since it is the container of the encrypted data that will be exposed to the users. The goal of this object is to provide a serializable/deserializable abstract class that can work with all the crypto workflow of HCL to contain the encrypted data.
The different implementations of this class will include:

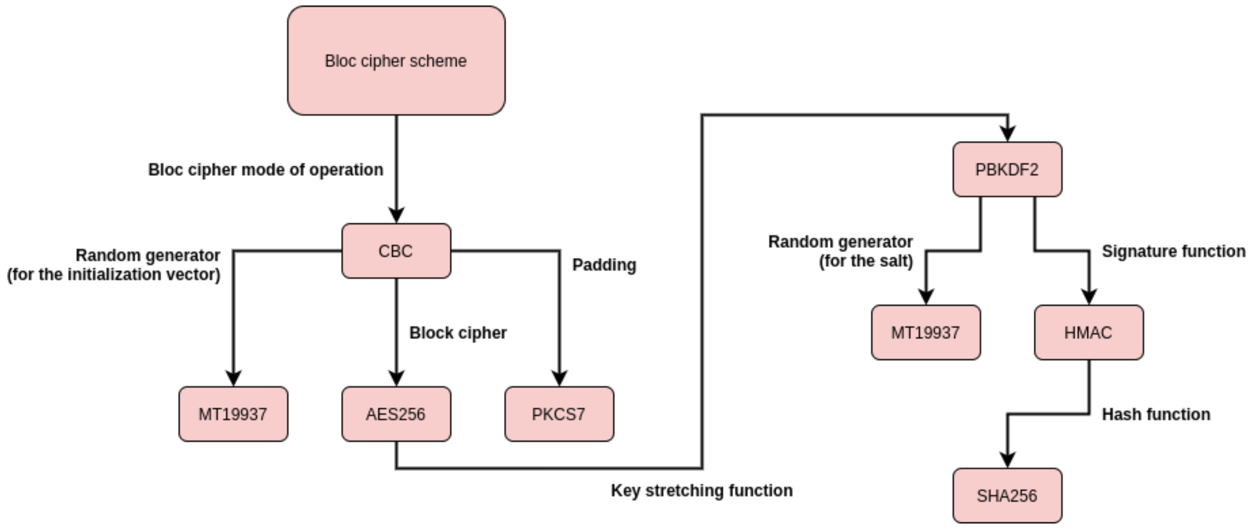
- ➔ Objects directly accessed and edited by the users:
    - ◆ Simple password (with a login or email, an associated service, and a name to retrieve it easily)

    - Textual note
    - Credit card
    - etc...
- Objects used internally by Harpokrat, mainly for the password sharing feature (more detail on their use will be detailed in the Password sharing section):
    - Symmetric key
      Will be used to encrypt shared vaults and itself encrypted using an asymmetric algorithm
    - RSA public key
      To encrypt data for the owner of the associated private key or verify the signature of a message the owner signed using the associated private key
    - RSA private key
      To decrypt data encrypted by the associated public key or to sign a message

Some other implementations may be added later.

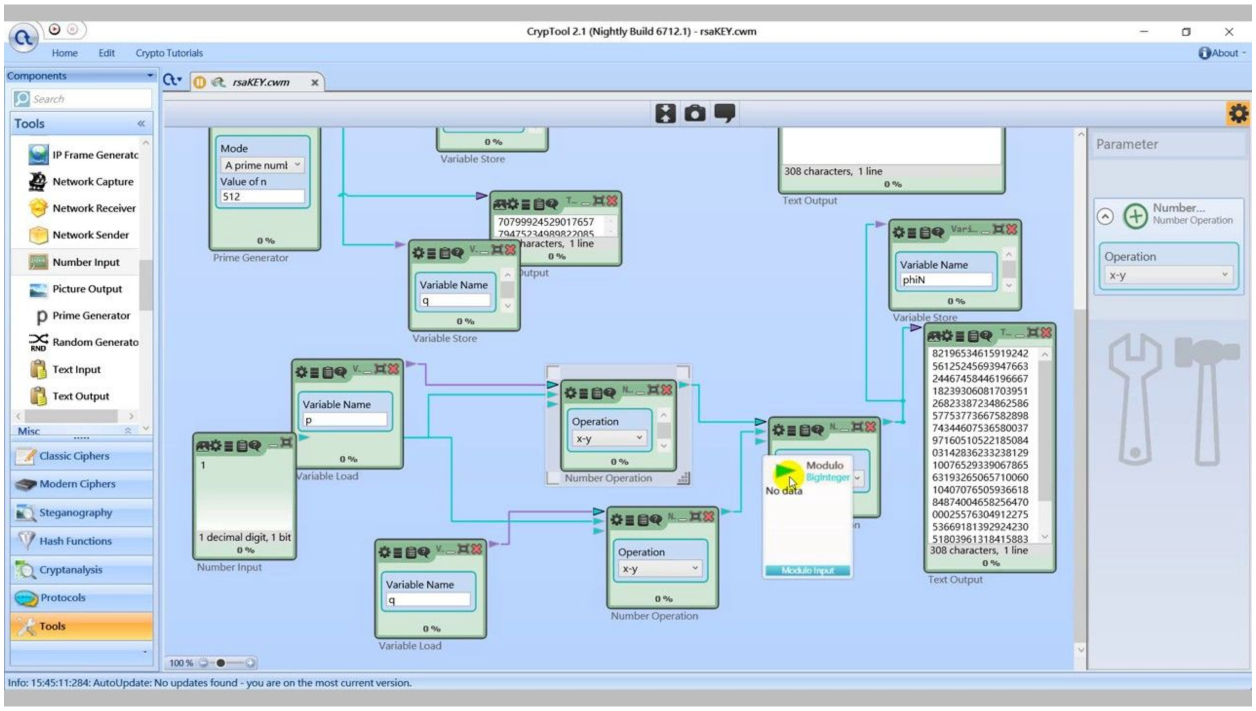
Crypto workflow

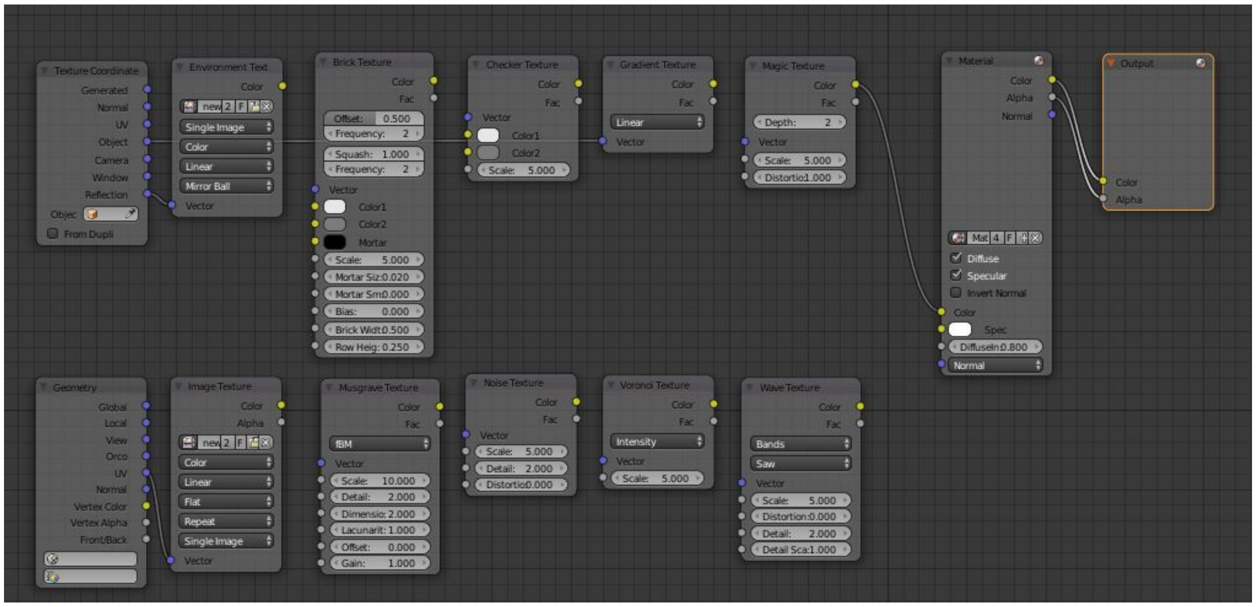
Overview



Example of a crypto workflow for a classic AES256 encryption scheme

The crypto workflow part of HCL is an attempt to have the cleanest architecture possible so that it is possible to implement numerous different cryptographic algorithms and connect them together in a flexible way. The inspiration for this feature was a mix of Cryptool and the node editor of Blender.


Cryptool interface with different “blocks” connected together


Blender node editor interface

In practice, all the blocks that compose a cryptographic scheme or workflow are implementations of a specific abstract class called `HCL::Crypto::ACryptoElement` and those classes are interdependent. In the illustration of an example of a classical AES256 implementation, we can see the dependencies between different `HCL::Crypto::ACryptoElement`. Each crypto element is also an implementation of a specific

element type like for example a random generator, a hash function, a key stretching function, etc…
This architecture allows us to create a virtually unlimited number of schemes by combining the implemented algorithms. The goal is to develop, in the future, an interface that would allow the technical team of a company using Harpokrat to customize their cryptographic schemes and maybe use internally developed crypto elements by providing a simple extension or plugin API.

In depth

Let us now talk a bit about how this crypto element based architecture works. The first thing to note is that we wanted this architecture to be very simple to use either when using it to create schemes by combining the crypto elements but also to add new crypto elements.

Dependency system

The `HCL::Crypto::ACryptoElement` is the basic building block of a crypto scheme. It has a type name, for example `"random-generator"` or `"key-stretching-function"`, it also has a name, for example `"mt19937"` or `"pbkdf2"` and finally it also has an id which is unique in the same type category of crypto elements. These three values allow the crypto element to be identified, for example to be displayed to an user in a node editor and to instantiate them easily via a method that we will detail later in this document.
Four very important member methods are defined in every crypto element:

```
const std::vector<std::string> &GetDependenciesTypes();
void SetDependency(std::unique_ptr<ACryptoElement> dependency, size_t
index);
void InstantiateDependency(const std::string &name, size_t index);
ACryptoElement &GetDependency(size_t index);
```

The method `HCL::Crypto::ACryptoElement::GetDependenciesTypes` returns a vector of all the dependencies types so an user can see which blocks this block depends on to be able to add them dynamically. This is basically what will define the inputs of a node in a node editor. To take a simple example, in the illustration of a classical AES256 implementation scheme, calling this method on the CBC block would return a vector containing: `"block-cipher"`, `"padding"` and `"random-generator"`.

The method `HCL::Crypto::ACryptoElement::SetDependency` is used to set a dependency of the current crypto element. The index corresponds to the index of the dependency in the vector of types returned by the previous method `GetDependenciesTypes`. The smart pointer to the crypto element must obviously be an implementation of the type identified by the string associated with the dependency index.

The method `HCL::Crypto::ACryptoElement::InstanciateDependency` allows us to directly create and set a dependency without having to instantiate it manually. Obviously the type of the dependency doesn't need to be specified since there is only one possible crypto element type for a specific index. The name corresponds to the name of the crypto element to instantiate which is of the type associated with the index.

And finally the method `HCL::Crypto::ACryptoElement::GetDependency` basically returns a reference to the dependency if it was set (and throws otherwise). This would allow to "load" an existing cryptographic scheme to display it to the user, modify an existing cryptographic scheme or even set dependencies recursively when using the method `InstanciateDependency`.

Here is an example of how to use these methods to implement the AES256 scheme from the example illustration:

```
HCL::Crypto::BlockCipherScheme cipher;
cipher.GetDependenciesTypes(); // {"block-cipher-mode"}
cipher.InstantiateDependency("cbc", 0);
HCL::Crypto::ACryptoElement &cbc = cipher.GetDependency(0);
cbc.GetDependenciesTypes(); // {"block-cipher", "padding",
"random-generator"}
cbc.InstantiateDependency("aes256", 0);
cbc.InstantiateDependency("pkcs7", 1);
cbc.InstantiateDependency("mt19937", 2);
HCL::Crypto::ACryptoElement &aes256 = cbc.GetDependency(0);
cbc.GetDependenciesTypes(); // {"key-stretching-function"}
aes256.InstantiateDependency("pbkdf2", 0);
HCL::Crypto::ACryptoElement &pbkdf2 = aes256.GetDependency(0);
pbkdf2.GetDependenciesTypes(); // {"message-authentication-code",
"random-generator"}
pbkdf2.InstantiateDependency("hmac", 0);
pbkdf2.InstantiateDependency("mt19937", 1);
HCL::Crypto::ACryptoElement &hmac = pbkdf2.GetDependency(0);
hmac.GetDependenciesTypes(); // {"hash-function"}
hmac.InstantiateDependency("sha256", 0);
```

After this block of code, the `cipher` object is ready to be used. Obviously, in a real world scenario all of this code will be dynamically called and interaction with the client will be implemented to allow the user to customize its scheme (even if we want to provide classical schemes by default since not all users will want to change their encryption schemes).

Automatic typed factory registering

This part of the program is extremely complex and requires at least a high level of c++ development knowledge. I will however try to simplify everything in this documentation as much as I can so it can be understood by most.

Basically, our objectives are:

➔ Easily instantiate any `HCL::Crypto::ACryptoElement` from anywhere in the code just by using two strings: one for the element type and one for the exact element name in the type category.

➔ We want to be able to implement new crypto elements and crypto element types easily without having to add some kind of manual instantiation or registration to the factories. In the same way we don't want to have to create new factories for every new crypto element type.

➔ The library must be very efficient and instantiation of crypto elements must be very fast since a lot can be required in a small amount of time when loading `HCL::ASecret` lists.

➔ And finally we want a very clean and maintainable code that we can be very proud of.

So to start with we will need a factory for every crypto element type and a factory of such crypto element type factories.

Let us start by talking about the factory of crypto element type factories since it is probably the simplest building block of this system. This class is called `HCL::Crypto::SuperFactory` and is a very simple implementation of a factory design pattern. Here is the very simplified class prototype:

```
namespace HCL::Crypto {
class SuperFactory {
 public:
  static bool Register(const std::string &, FactoryInstanceGetter);
  static AFactory &GetFactoryOfType(const std::string &);
  static const std::vector<std::string> &GetFactoryTypes();
};
}
```

As you can see there is nothing specific about this SuperFactory which is just a simple static factory to instantiate `HCL::Crypto::AFactory` objects.

We will now look over this `HCL::Crypto::AFactory` class. Contrary to what its name implies, an AFactory is an interface and not an abstract. Here is the simplified prototype of this interface:

```
namespace HCL::Crypto {
class AFactory {
 public:
  std::unique_ptr<ACryptoElement> BuildFromHeader(const std::string
&header, size_t &header_length);
  std::unique_ptr<ACryptoElement> BuildFromId(uint16_t id);
  std::unique_ptr<ACryptoElement> BuildFromName(const std::string &name);
  const std::string &GetFactoryType() const;
  const std::map<std::string, uint16_t> &GetClassesNames();
};
}
```

This prototype is still very simple and would be sufficient for this system on its own if we wanted to manually create factories for every crypto element type. However, one of our objectives is to simplify as much as possible the work of the developer so more crypto elements and crypto elements types can be created without the pain of all the setup, instantiation, registration, factory implementation, etc… and this is the reason why we created the `HCL::Crypto::Factory` templated class.

This class could be seen as a simple templated factory. However, because we didn't want to reference them and register them to the super factory in some kind of initialization function, we had to trick the c++ compiler to make it make the factories register themselves automatically.
To do so we defined a private static member to the templated class definition (which, in c++ is one static value for each different value of the template argument) and initialized it with the result of the super factory registration method. Here is a very naive implementation of such an auto registration system:

```
namespace HCL::Crypto {
template<typename AbstractClass>
class Factory : public AFactory {
 public:
  static AFactory &GetInstance() {
    static Factory<AbstractClass> instance;
    return instance;
  }
 private:
  static const bool is_registered_;
};

template<typename AbstractClass>
const bool Factory<AbstractClass>::is_registered_ =
SuperFactory::Register(AbstractClass::GetName(), GetInstance);
}
```

As you can see, if a reference to the templated class is used anywhere in the sources of HCL, the compiler will create a definition of this class for the template argument thus initializing the `static const bool is_registered_` member with the result of the `HCL::Crypto::SuperFactory::Register` method, thus calling it and registering the defined factory in the super factory. However, as you will discover it later, we don't even have to use a reference to the templated class directly thanks to the `HCL::Crypto::AutoRegisterer` class.
This factory implementation also implements the `AFactory` `BuildFromHeader` method which allows the creation of a full cryptographic workflow or scheme from a serialized `HCL::Crypto::EncryptedBlob` object. I won't enter in too much details on how this object works and how the encryption scheme is serialized but it is basically the concatenation of the crypto elements ids and of some parameters specific to each crypto element implementation like random generated vectors and such. The deserialization is basically a recursive parsing of serialized headers containing those ids and parameters.

The last very important class is the `HCL::Crypto::AutoRegisterer`. This class uses the same trick as the `Factory` class to get auto-registered but instead of to the `SuperFactory`, it gets registered to the `Factory` class of its type. To do so, the AutoRegisterer class has two template parameters, representing the type class and the implementation class. Here is a simplification of the implementation of this class:

```
namespace HCL::Crypto {
template<typename AbstractClass, typename RegisteredClass>
class AutoRegisterer : virtual public AbstractClass {
 public:
  static std::unique_ptr<AbstractClass> Instantiate();
  static uint16_t GetId() {
   return RegisteredClass::id;
  }
  static const std::string &GetRegisteredName() {
    return RegisteredClass::GetName();
  }
  static const std::string &GetTypeName() {
    return AbstractClass::GetName();
  }
 protected:
  static const bool is_registered_;
};
template<typename AbstractClass, typename RegisteredClass>
const bool AutoRegisterer<AbstractClass, RegisteredClass>::is_registered_ =
    Factory<AbstractClass>::Register(RegisteredClass::id,
                                     &AutoRegisterer<AbstractClass,
RegisteredClass>::Instantiate,
                                     GetRegisteredName());
}
```

This class basically allows us to create new crypto elements very easily by just inheriting this class and setting the appropriate template parameters. This is also thanks to this class that an actual reference to the templated factory is used and that is how the trick used to automatically register those factories works.

Let us now imagine that a developer contributing to the HCL wants to add a whole new crypto element type and an implementation of an algorithm.
The first step would be to create a folder of the name of the type (plural) in the `src/services/Crypto` directory; this is absolutely not a technical requirement but will help to understand and structure the source files architecture better.
The second step is the creation of an abstract class representing the crypto element type. This crypto element type must obviously inherit from the `ACryptoElement` class. It is in this abstract class that you can define type-specific methods used by the other crypto elements using this crypto element type as a dependency.
The third and final step is the creation of a class which is an implementation of the crypto element type which will inherit from the `AutoRegisterer` class with the type class as a first template argument and the implementation class type as the second template argument.

Let us take an example of the creation of a new crypto element type and a first algorithm implementation. Let us take as the example the creation of a random generator type and, as a first implementation let us create a class implementing the Mersenne Twister 19937 algorithm.
For the first step we will create the folder `src/services/Crypto/RandomGenerators`.
For the second step we will create an `ARandomGenerator` class (simplified):

```
// src/services/Crypto/RandomGenerators/ARandomGenerator.h
namespace HCL::Crypto {
class ARandomGenerator : public ACryptoElement {
 public:
  virtual uint8_t GenerateRandomByte() = 0;
  static const std::string &GetName() {
    static std::string name = "random-generator";
    return name;
  };
};
}
```

As you can see in this implementation, we created a pure virtual method `GenerateRandomByte` which will be implemented by any implementation of a random generator class. You can also note the presence of the implementation of the GetName method which will allow the `SuperFactory` to create factories creating this type by name. There is no need for an id since the serialization doesn't need to precise the type of the serialized crypto element as this information is known by the crypto element possessing this dependency.
The third step for our example is to create the implementation of the class of this type using the Mersenne Twister 19937 algorithm. Here is the prototype of such an implementation (simplified):

```
// src/services/Crypto/RandomGenerators/MT19937.h
namespace HCL::Crypto {
class MT19937 : public AutoRegisterer<ARandomGenerator, MT19937> {
 public:
  MT19937();
  uint8_t GenerateRandomByte() override;
  const std::string &GetElementName() const override { return GetName(); };
  const std::string &GetElementTypeName() const override { return
GetTypeName(); };
  static const uint16_t id = 1;
  static const std::string &GetName() {
    static std::string name = "mt19937";
    return name;
  };
};
}
```

And voilà ! Thanks to the system of `SuperFactory`, `AFactory`, `Factory` and `AutoRegisterer`, those two (simplified) class definitions are enough to create a new crypto element type and a first implementation. Now the `Factory` of type `ARandomGenerator` is automatically created and registered to the `SuperFactory` and a simple code like the following is enough to instantiate the new crypto element:

```
auto mt19927_cbc =
HCL::Crypto::SuperFactory::GetFactoryOfType("random-generator").BuildFromNa
me("mt19937");
```

Now another crypto element could require a `"random-generator"` dependency and everything is already linked together automatically.

Algorithms

We will not enter in too much detail to explain how every implemented algorithm works since there is a lot of details online (mainly Wikipedia which is a great source of documentation) and the c++ implementation is pretty trivial for an experienced c++ developer. However, here is the list of currently implemented cryptographic or related to cryptography algorithms classified by type (more will come in the future):

- ➔ Block cipher modes
  Elements implementing the different modes of chaining of block ciphers for encryption of messages that have a size different than the cipher size but also to improve security by adding new notions like initialization vectors
  - ◆ CBC
  - ◆ ECB
- ➔ Block ciphers
  Basic symmetric block cipher algorithms implementations
  - ◆ Rijndael
    - AES128
    - AES192
    - AES256
- ➔ Ciphers
  This is the Harpokrat internal representation of the “type” of workflow that will follow
  - ◆ BlockCipherScheme
- ➔ Hash functions
  Simple cryptographic hash functions
  - ◆ SHA2
    - SHA224
    - SHA256
    - SHA384
    - SHA512
- ➔ Key stretching functions
  Algorithms used to extend the size of a key to a certain size, and most of the time improving its security exponentially
  - ◆ PBKDF2
- ➔ Message authentication codes
  This elements allows for the signing of a message to testify its integrity and authenticity
  - ◆ HMAC
- ➔ Padding
  Element to fill up a block of a certain size with different strategies of padding data
  - ◆ PKCS7

- Primality tests
  Elements to test the primality of numbers, developed mainly to be used by the RSA algorithm
  - Fermat
- Prime generators
  Generators of prime numbers, developed mainly to be used by the RSA algorithm
  - CustomPrimeGenerator
    This element is using the LibGMP. We explain the use of this library later in the documentation.
- Random generators
  Generators of random numbers or byte sequences of variable length
  - MT19937
- Unclassified algorithm
  These algorithms are not implemented as crypto elements but are, instead, used directly by other parts of the library since there is no need for flexibility because they cannot really be replaced by other “equivalent” algorithms
  - Base64
  - Rijndael key schedule
  - Rijndael substitution box

LibGMP

In order to deal with problems encountered with the management of big numbers, the installation of an external library was more than necessary. That is why the management of big numbers is now in the hands of the libGMP.

Indeed the libGMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine the library runs on. The libGMP has a rich set of functions, and the functions have a regular interface. In other words, it is mostly designed to be used in cryptography applications, security applications or algebra systems.

Its use greatly helps HCL by the fact that it implements many algorithms treating prime numbers as some primality tests. HCL thus gains in performance.

The link to their documentation: https://gmplib.org/manual/

Javascript Wrapper

The Javascript wrapper allows the use of the HCL library inside a browser based project Javascript or Typescript project. It is made in typescript and designed to have no dependency in order to make it as easy as possible to maintain and to control the entire codebase.

To use the wrapper you must instantiate a HclwService service.
You can then use various methods

- service.getBasicAuth(email, password): returns a promise of a string containing the basic auth to use when requesting the JWT token
- service.getDerivedKey(password): returns a promise of a string containing the derived key based on the password
- service.createSecret(key: string, content: string): returns a promise of a secret object that allows to build secrets and get the encrypted content to save on the server

Python Wrapper

The Python wrapper is a Python library that allows the use of the HCL in any project developed in Python. It makes use of the ctypes library to bind C exported functions from the system wide installed HCL library in Python.
The code of this wrapper is actually very simple. Let us look at the library loading:

```
hcl_library_name = ctypes.util.find_library('hcl')hcl_library =
ctypes.cdll.LoadLibrary(hcl_library_name)
```

As you can see, the use of the ctypes library is very simple and is consistent between all platforms.
Then to use a function we simply need to define its argument types and return type like so:

```
hcl_library.GetSecretFromContent.argtypes = [ctypes.c_char_p,
ctypes.c_char_p]
hcl_library.GetSecretFromContent.restype = ctypes.c_void_p
```

Then calling the functions is also very trivial:

```
Hcl_secret = hcl_library.GetSecretFromContent(wrapper.encode_string(key),
wrapper.encode_string(content))
```

For an easy and more “pythonic” use of the wrapper, we created binding classes that create a more business oriented use of the library. For example here is how a Secret object gets created (example from the python library, used by the CLI client):

```
from hclw.HCLW import HCLW
from hclw.Secret import Secret

Wrapper = HCLW()
secret = Secret(wrapper, key, content)
```

As you can see this simplifies a lot the use of the library and wraps the utility functions like the string / char array conversion functions.

Flutter Wrapper

The Flutter wrapper is a Flutter library that allows the use of the HCL in any project developed in Flutter. Flutter directly has a `DynamicLibrary` module that allows us to load dynamic libraries and wrap and use the exposed C functions in Flutter.
The code of this wrapper is actually very simple. Let us look at the library loading:

```
class HclwFlutter {
  DynamicLibrary _hcl;
  HclwFlutter() {
    this._hcl = Platform.isAndroid
        ? DynamicLibrary.open('libhcl.so')
        : DynamicLibrary.process();
  }
}
```

Then to use a function we simply need to define its argument types and return type like so:

```
typedef fncPtrFrmChrArr = Pointer<Void> Function(Pointer<Utf8>);
GetSecretFromContent =
this._hcl.lookup<NativeFunction<fncPtrFrmChrArr>>('GetSecretFromContent').a
sFunction<fncPtrFrmChrArr>();
```

Then calling the functions is also very trivial:

```
_hclSecret = GetSecretFromContent(Utf8.toUtf8(key), Utf8.toUtf8(content));
```

For an easy and more object oriented use of the wrapper, we created binding classes that create a more business oriented use of the library. For example here is how a Secret object gets created (example from the Flutter client):

```
import 'package:hclw_flutter/secret.dart' as hclw_secret;
decryptedSecret = new hclw_secret.Secret(
            lib, content: encryptedSecret, key: user.password);
```

As you can see this simplifies a lot the use of the library and wraps all the Flutter specific wrapping systems which would not be very pleasant to use directly in the Flutter client.

Api Client Libraries

Javascript Client Library

Dependencies

The javascript library is designed to be as light and flexible as possible, it's only dependency is the harpokrat cryptographic library, used to handle secret encryption.

Architecture

The library is written in typescript (TS) and is compiled into javascript (JS) + TS definition files for production using webpack.
It is composed of a main class called HarpokratApi that has several members representing each endpoint. The goal was to make the code required to use the library as easy and clear as possible, for example to create a secret we would do api.secrets.create(...).

Endpoints

Each endpoint consists of an interface and an implementation (Ex: ISecretEndpoint implemented by SecretEndpoint), only the interface is exported and is visible when using the library.
All endpoints implement the IEndpoint interface which exposes the path of the root of the endpoint on the remote server (Ex: “/secrets” for the Secrets endpoint).
Most endpoints implement the IResourceEndpoint interface, this interface has all the CRUD methods for a given resource, this allows to reuse the code through a generic ResourceEndpoint implementation that is then subclassed by most endpoints.
The only exception is the JsonWebToken endpoint which only allows for the Create operation, as it is used for authentication and is a pseudo-resource.

JSON Api

The remote api follows the JSON Api guidelines, interfaces for each object of the format have been defined through typescript interfaces, this allows to simply parse the JSON of the response as a series of JSON objects without instantiating new classes but also provide the strong typing of TS.

Resources

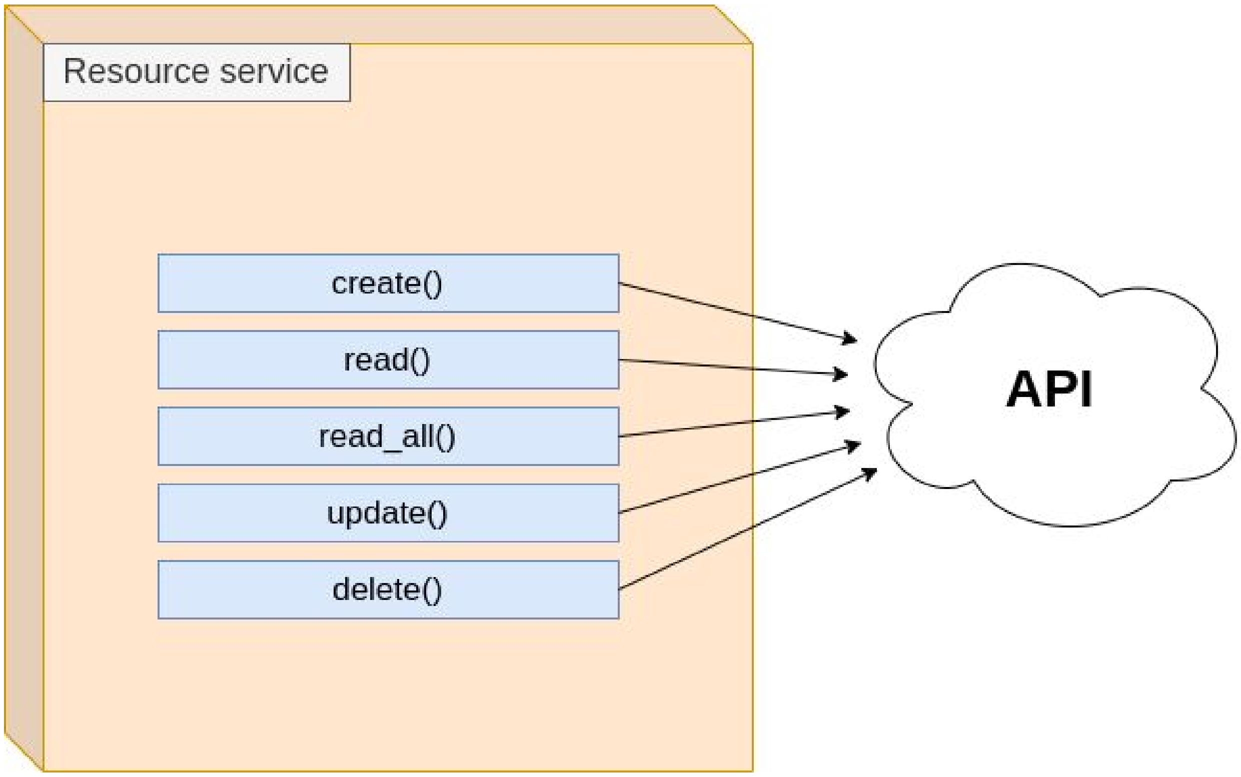
Resources are also TS interfaces that describe the attributes of each resource, there is also an alias type defined for each resource that defines the non-generic JSON Api resource. This allows to make the code smoother and easier to read, for example instead of writing “const secret: IResource<ISecret>”, we can write “const secret: ISecretResource”.
It is important to note that ISecret and ISecretResource are different but the former is included in the latter as the “attributes” property.

Requester

In an effort to make the library usable with as many frameworks as possible, the main Api class has a requester option, this is passed to every endpoint and is used to make HTTP requests to the remote server.
A valid requester must implement the IRequester interface which can only be implemented by a function since it must be callable. It accepts a URL and various options (HTTP method, body, headers, query params and auth).
Auth is simply a boolean indicating whether or not to include the auth header which is provided by the library automatically.
By default the requester is implemented using fetch as it is a native function available in most modern browsers.

Python Client Library

The python library allows you to link python code (such as the python CLI), the API and the python wrapper which allows you to use the HCL encryption library.



An illustration of the general functioning of the resource service.

Its architecture is a layer of the javascript library, a module containing different services is instantiated, each of these services corresponds to a functionality of the API and will have methods that will allow it to execute different actions with the API and HCL. The library also contains Harpokrat related business objects such as Secrets or Users which allows python developers to be more efficient.

To install the library dependencies :

```
pip3 install -r requirements.txt
```

To install the library within a project :

```
pip3 install {{path to python library}}
```

Angular Library

Architecture

The angular library is made in TypeScript (TS) for the Angular framework, it is a collection of services and components that allow to streamline the production of web clients and web extensions. It uses the Harpokrat Cryptographic Library along with the Harpokrat Javascript Library to interact with the remote server.