Real-time GNU Taler auditor

Bachelor's Thesis

Real-Time GNU Taler Auditor

Course of study	Bachelor of Science in Computer Science
Author	Cédric Vincenz Zwahlen and Nicola Sacha Eigel
Advisor	Prof. Dr. Christian Grothoff
Co-advisor	Prof. Dr. Emmanuel Benoist
Expert	Han van der Kleij

Version 0.1 of June 13, 2024

- Technik und Informatik
- Mikro- und Medizintechnik

Abstract

One of the key components of the GNU Taler payment system is the auditor, which is used to ensure that a payment service provider operating the payment system is operating correctly. The primary goal, is to provide assurances against insider threats, compromised systems or data corruption due to technical failures.

In the context of this thesis, the GNU Taler auditor was improved, and now works in real-time, thus providing operators and regulators with more timely insights into the payment system. This was achieved by changing the existing logic, which would previously generate periodic JSON reports, to a database-centric approach. By implementing a REST API service for the newly generated database tables, the newly created single page application is able to visualize audit data in real-time on its dashboards.

To achieve those changes, the six GNU Taler auditor helper programs, each responsible for analyzing different parts of a GNU Taler exchange, were adapted. The existing report generating logic was analyzed and the database was extended with tables to store the various findings generated by the auditor. This replaces the existing periodic report generating logic.

The new tables contain distinct aspects of GNU Taler that are relevant to the auditing process, such as failures, delays in processing, active operations, or simply the system state with the amounts of currency in circulation or the total amount of the various payment fees earned by the exchange. For each of the new tables, new REST API endpoints were designed, documented and implemented.

This enabled the development of a new auditor frontend, the single page application for displaying the data in an easy, understandable and digestable manner. Necessary access control precautions were taken into consideration and implemented.

To foster sustainable development practices, the auditors unit tests were also adapted and changed. Due to the database-centric approach, the unit tests now not only need tests for the main auditing logic, but also tests for the functionality of the REST API. Each test case begins by running the auditor helpers, which insert various reports into the database. After a fault injection, the tests then query the database via the REST API and then check that the correct findings are returned by the REST API.

Acknowledgements

We want to sincerely thank Christian Grothoff and Emmanuel Benoist for the opportunity to work on and improve GNU Taler. Their teaching, support and trust was immensely helpful to this work and our growth.

We also want to thank the whole GNU Taler team, especially to Özgür Kesim, Florian Dold and Sebastian Marchano for their knowledge and technical support.

We also want to thank our families and friends for enduring us, their support, time, understanding and accommodation.

Lastly, we want to acknowledge all the previous contributions to the auditor and GNU Taler in general. Big parts of the code and the logic from the auditor and associated components were already existing and we merely extended or adjusted it to make it real-time or fit its cause. Also, a lot of documents, texts and explanations, which we used in this work.

Contents

1. Introduction

1.1. Motivation

The world of money is in a state of change. Nations all over the world are embracing the research and development of new kinds of payment systems or currencies themselves. The call for a faster transaction finality, better ease of use and the removal of middlemen, while preserving or amplifying the security and privacy of payment systems, is supported by the further development of the Internet and society.

Throughout the history of money, which led to this current need for change, people tried to abuse the available payment systems to their advantage. Resulting in disastrous losses for involved people and societies, like Bernie Madoff's ponzi scheme [1] or Wirecard's "questionable" accounting practices [2]. The lack of accounting and auditability is prevalent, seemingly no payment system exists that includes a secure by design approach. The newest technology hype, cryptocurrencies and central bank digital currencies, often building on blockchain technology, may have found a solution for distributed digital payments. However, these contemporary systems are inadequate to stop the next meltdown, as evidenced by the fact that, they are more often than not, key to enabling criminal activities. [3] [4]

With the emerging possibility of online micro payments creating potentially a 10–1000x increase in transaction volume with virtually instant transaction finality, security and privacy requirements for modern payment systems are higher then ever. Our objective is to show a way to cope with those requirements and to create a building block for the payment system of the future.

Our work focuses on the GNU Taler payment system, which differs from most existing payment systems by its comprehensive use of digital signatures. This work enhances the auditor system of Taler, which is designed to detect and mitigate operational problems to prevent financial tragedies. The auditor is an independent component which can be attached to a Taler payment service provider (an exchange) and then monitors Taler's internal transactions as well as transactions in the core banking systems. The goal is to verify that the exchange is operating correctly. This provides the much needed ability to amplify the security and auditability of a payment system to prevent fraud, insider threats and other shortcomings.

1.2. GNU Taler

Figure 1.1.: Logo of GNU Taler [5]

GNU Taler is a payment system that provides a way to pay digitally and anonymously. It is built on the following design principles:

1. Free/Libre Software
2. Protect the privacy of buyers
3. Auditability - enable the state to tax income and crack down on illegal business activities
4. Prevent payment fraud
5. Collect the minimum information necessary
6. Be usable
7. Be efficient
8. Fault-tolerant design
9. Foster competition

Its goal is to be like cash, but digital. This means providing near instant transaction finality, the ability to do micro payments and it has to be easy to use. It's currently the only payment system worldwide that manages to protect the buyers privacy, while simultaneously enabling the state to enforce tax on merchants and allow for the implementation of anti money laundering (AML) logic.

GNU Taler is neither based on a blockchain, nor based on some other type of decentralised ledger; instead, payment services are offered by providers that use a traditional SQL database. To provide cash-like privacy for payers, it uses a concept called blind signatures [6] and advanced state-of-the-art cryptography.

1.3. Real-Time GNU Taler Auditor

Conscientious and thorough auditing is vital for any serious payment system and the assumption it's useless or unnecessary is beyond naive and will inevitably lead to disaster. Cases like the previously mentionend Wirecard fraud make it clear that there is a real need for automated systems to verify the integrity of payment services. This is exactly what a GNU Taler auditor does.

In fact, precisely because GNU Taler is intended to be a micro payment service where transactions take milliseconds to complete and is expected to handle hundreds of small payments a minute, it must provide an automated auditor. Because auditing such a magniture of transactions by hand to find discrepancies or patterns of misbehaviour, would simply be impossible. And even if it was possible to comb through all this data manually, because the tokens are blinded, establishing a trail between them is futile.

From the beginning, GNU Taler was designed to include auditor capabilities. The auditor is not only designed to give peace of mind for the developers, but also for its operators, users and regulators.

The auditor is a core component of the GNU Taler. It receives a replica of the exchange's database as its primary source of truth. Additionally, the auditor must have access to the core banking system to inspect the wire transfers of the exchange's bank account, and it also receives input from merchants.[1] The auditing logic is implemented in six helper programs which verify that the state of the various databases is consistent. The auditor generates reports that summarize the state of the system and detail various discrepancies, with the goal of identifying attacks from both outsiders and insiders.

[1] In the future, it should also receive inputs directly from wallets.

1.4. Goals

The goal of this work, was to adapt the auditor, so it could present its findings in real-time.

The existing auditor logic, primarily the six helper programs, was to be extended and to store results in a database instead of in JSON files. Thus, part of the work involved extending the database schema. Database triggers and the LISTEN-NOTIFY mechanism of PostgreSQL [7] would need to be added to notify helper programs of new database records, activating the respective helper logic instantly instead of relying on periodic jobs.

The resulting audit data should then be made accessible via a REST API, allowing it to be queried and displayed in an easy fashion. For this, a single page application should show inconsistencies in the exchange as they are discovered. The data should be organized into various dashboards that are easy digestible and user-friendly. The webpage should also allow operators to silence warnings they have already investigated, allowing operators to keep track only of still relevant information.

1.5. Scope

The scope of this work is as follows:

- Renovate the current auditors logic, its six helper programs, to store its report data in a database instead of generating reports. Abide by existing GNU Taler design and coding standards.
- Extending the auditors REST API logic, to provide the ability for retrieval of parts of the auditors database. Adding a protection layer is obligatory. The REST API design has to be well documented in the official GNU Taler docs.
- Creating a single page application, with some reasonable form of access control protection, that displays the highly valued information in an easy digestible and understandable manner. The application shall be built on the same technology as existing GNU Taler backends for other components.
- Adjust the existing auditor unit tests to work with the new auditors database-centric structure.

2. Preliminaries

2.1. GNU Taler actors

A digital transaction always features at least two actors; the customer and merchant. GNU Taler needs two more – the exchange and the wallet. Each actor has its own set of responsibilities and capabilities. The auditor's task is to implement verification mechanisms, to audit each of these actors.

2.1.1. The Exchange

The Taler exchange is run by a bank – it may even be run by a central bank. Its main function is to exchange 'regular' currency – say swiss francs – into GNU Taler currency, one unit of which, is called a token. A given amount of francs, may be exchanged for an equivalent amount of GNU Taler of the same value. So, exchanging 1 CHF, yields a token worth 1 CHF. This means, that GNU Taler is not a separate currency, but simply an alternative way to spend money, that is digital and anonymous.

2.1.2. The Wallet

Tokens received from an exchange are stored in a GNU Taler wallet, which may be installed on phones or opened in browsers. From there, tokens may be spent directly at merchants or be sent to other people. An exchange does not know which tokens it has issued to whom. While this is great for privacy, it also means that anyone in possession of GNU Taler tokens is solely responsible for keeping them safe – a lost token may not be recovered and replaced. To spend tokens, an internet connection is required. During peer to peer payments, while it may seem like users can pay someone directly, tokens are actually first sent to the exchange, and then, a newly anonymized token is passed along to the recipient.

2.1.3. The Merchant

Tokens may be spent at any merchant that accepts them. A merchant can then contact an exchange, to redeem any tokens it received and have their equivalent value be deposited in their bank account. This system ensures, that a payers identity remains anonymous, while merchants must disclose themselves to an exchange to receive money. This important for tax purposes.

2.1.4. The Auditor

While the auditor is an important part of GNU Taler, it does not issue, redeem or receive currency. Instead, it constantly monitors an exchange's database, and verifies its soundness. An exchange verifies tokens it receives and vets merchants, and auditors make sure an exchange acts as expected. A compromised exchange could generate huge losses for its operators, which makes auditors that may detect discrepancies early, essential to GNU Taler's security.

The auditor's role is to find misbehaviors or fraud attempts and monitoring the systems status. Those can be technological problems like network failures or system downtimes or things like database manipulation or other issues.

GNU Taler is also equipped to deal with insider threats. Ideally, several instances of the GNU Taler auditor auditing the same exchange are run simultaneously, in different physical locations, by different organizations. This way, even if one auditor is manipulated, others can still operate correctly.

2.2. GNU Taler Architecture

To understand the auditor and the exchange, one needs to understand Taler's payment flow, its concepts and the structure.

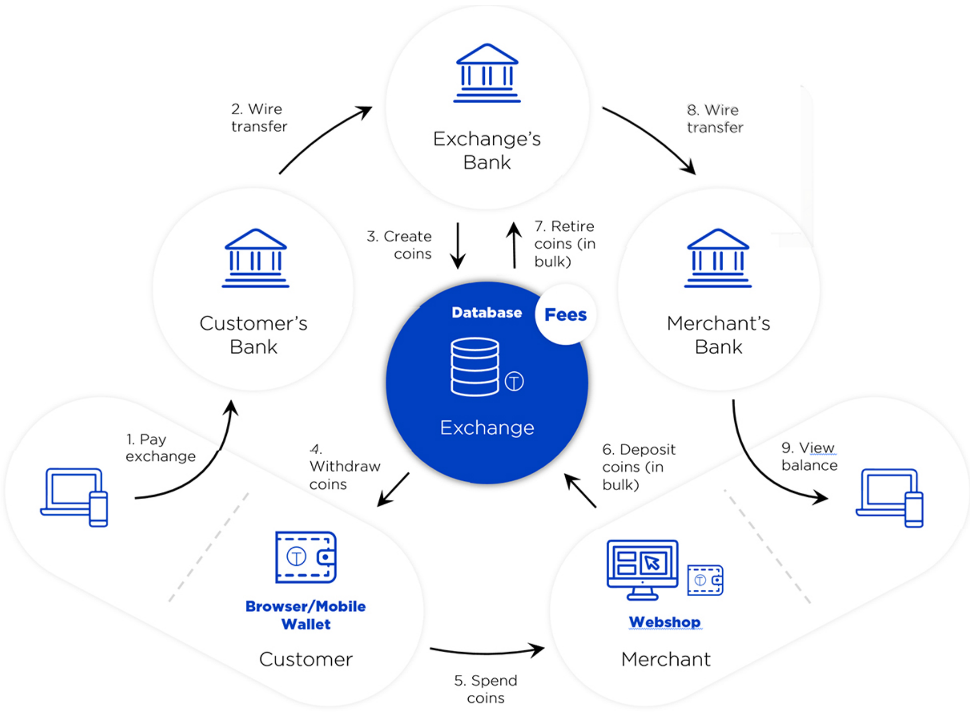

Figure 2.1.: Overview of the taler architecture [8]

The figure 2.1. shows an overview of Taler's architecture and the payment flow; which works as follows:

1. The customer sets up a wire transfer through its bank of choice.
2. The bank operating the exchange receives the wire transfer.
3. The operating bank then creates the tokenized coins, in the amount of the received value of the wire transfer. Those coins are blinded using a blind signature (see chapter 2.3.3.), meaning, the exchange does not know (!), who redeems these coins.
4. The customer redeems those coins through his wallet anonymously.
5. The customer spends his coins by buying something or sending money to another party.

6. As soon as the merchant wants to redeem the money to have it in his bank account, he deposits his coins to the exchange. He does this in bulk, meaning, a whole stack of coins out of transactions.
7. The exchange passes received coin deposits to the operating bank.
8. The exchange operating bank sends a wire transfer to the merchant's bank.
9. The merchant receives the money.

2.3. GNU Taler Concepts

There are some unique problems a digital payment system needs to master. The concepts, methods and systems that solve them and lay the essential groundwork for the GNU Taler payment protocol are elaborated in this chapter.

Important concepts needed to understand the auditor, are:

2.3.1. Coins and Denominations

There may be a product on sale for 42 swiss francs. To buy it, a wallet needs to have coins in the value of at least that amount plus the transaction fees. Say the wallet has ten coins with a value of 5, it would pay with 9 out of those ten coins, to a total of 45 swiss francs. Say the merchant wallet now only has coins with value of 5 swiss francs as well, it could not return change properly. That's were denominations come into play. Coin denominations represent values of a coin, say 50 centimes. The payment can be concluded by paying with coin denominations.

2.3.2. Keys and Signatures

Taler uses cryptography to ensure it can hold what it promises. One cryptographic system used throughout is public-key cryptography [9]. This system uses pairs of keys called public and private keys. Such key pairs are used whenever two actor communicate with each other via the internet.

2.3.3. Blind Signatures

Another cryptographic system, absolutely essential to GNU Taler, are blind signatures [10]. Their goal is to provide unlinkability and anonymity to coins, and thus making it impossible for the exchange to identify the customer redeeming them. Blind signatures can be understood as an extension of public-key cryptography. It functions like a ballot that has been put into an envelope. The envelope then gets signed by the authority, but the authority does not know what is inside the envelope. Similarly, the exchange does not know whom it issues the coins to, but knows they are valid because it signed them.

2.3.4. Wire Transfer

A wire transfer is simply a money transfer between a bank and its exchange. A wire transfer is accompanied by a transfer fee.

2.3.5. Purse

A purse is used in a peer to peer transaction. The payer can put their coins into the purse, which expects a previously determined sum of money, and the payee may redeem the coins in the purse once the payer put the required amount into it. A purse is managed by the exchange. [11]

A purse can expire, either because the payer fails to fill it with enough coins or because the payee does not claim their money.

2.3.6. Reserves

When the customer pays the exchange's operating bank to receive some GNU Taler, the exchange opens up a reserve. The customer can then withdraw his money from the newly created reserve into his wallet. If a reserve is not emptied, the exchange will eventually close it.

2.3.7. Revocation

A revocation in the context of a signature means, that a signature is declared invalid. If the signature is still used to sign something, the validation will fail, because a signature validation process includes querying a signature's revocation status.

2.3.8. Recoup

Operations by which an exchange returns the value of coins to their owner, because their signature is no longer valid. Either, the exchange allows the coin's owners to withdraw new coins with a valid signature, or it wires funds back to the bank account of the coin owner.

2.3.9. Dirty Coin

A coin is dirty if its public key may be known to an entity other than the customer, thereby creating a situation, in which some entity might be able to link multiple transactions of coin's owner if the coin is not refreshed.

2.3.10. Melt

Melting is a step of the refresh protocol. It includes invalidating a dirty coin to then be renewed in a subsequent reveal step.

2.3.11. Refresh

Operation by which a dirty coin is converted into one or more fresh coins. Involves melting the dirty coins and then revealing so-called transfer keys.

2.3.12. Reveal

A step in the refresh protocol where some of the transfer private keys are revealed to prove honest behavior on the part of the wallet. In the reveal step, the exchange returns the signed, fresh coins.

2.4. Auditor architecture

The exchange stores a lot of information to function properly. Including balances, wire transfers, completed transactions, as well as such still in flight.

However, the exchange is not the auditor's only source of information, it also receives data from the merchant and the banking system interface software LibEuFin. [12] It's these different sources of information, that makes this auditor so powerful.

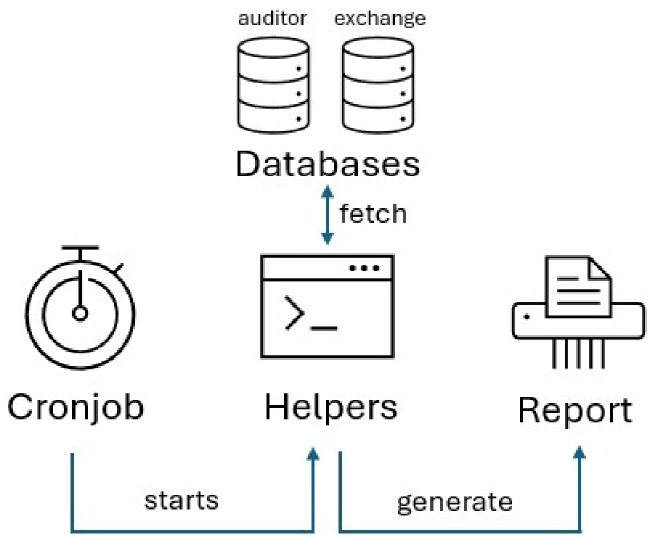

Figure 2.2.: Old Auditor architecture simplified

Previously, the auditor ran alongside the exchange, where it was configured to run as a periodic running task (cronjob). The helpers then recorded all discrepancies as they were found, and generated JSON report files before shutting down. This is not ideal, because the results of an audit can only be seen after the fact. Also, an auditor may run for a long time, so any results that are found may accumulate over a long period of time – without being seen.

This not only makes it more difficult to mitigate the cause of the problems found, but might also be overwhelming for any person that would have to review these audits.

2.5. Protocols and States

In this chapter, select protocols of GNU Taler are explained. They give an idea of some of the exchange's processes. Understanding these concepts somewhat will be useful for further chapters.

2.5.1. Reserve

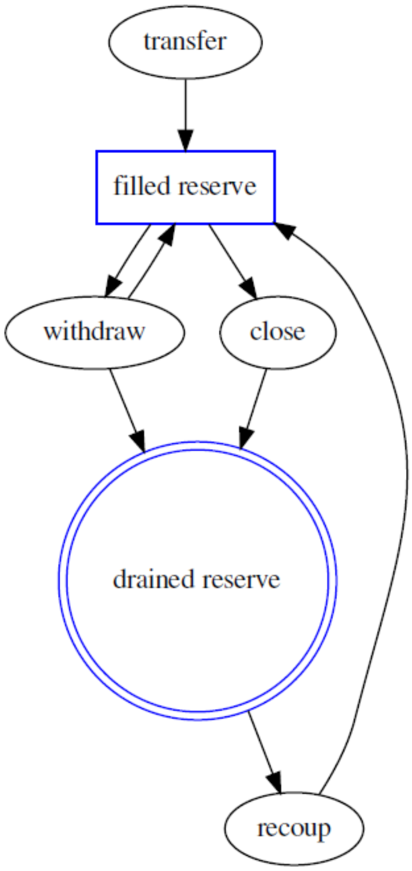

Figure 2.3.: Overview of states and state changes of the reserve [13]

A user obtains GNU Taler, by asking his bank to wire some money to an exchange. This initiates a wire transfer from the bank to the exchange. The exchange then creates a reserve, filled with coins worth the same as the money payed to the bank, minus fees. The user is given the private keys to the reserve, and can withdraw those coins. This drains the reserve and leads to the "drained reserve" state, once all funds are withdrawn. The reserve itself closes after a certain time, even if it is not fully drained. A recoup operation is then possible, which will lead to a filled reserve state again.

2.5.2. Coin

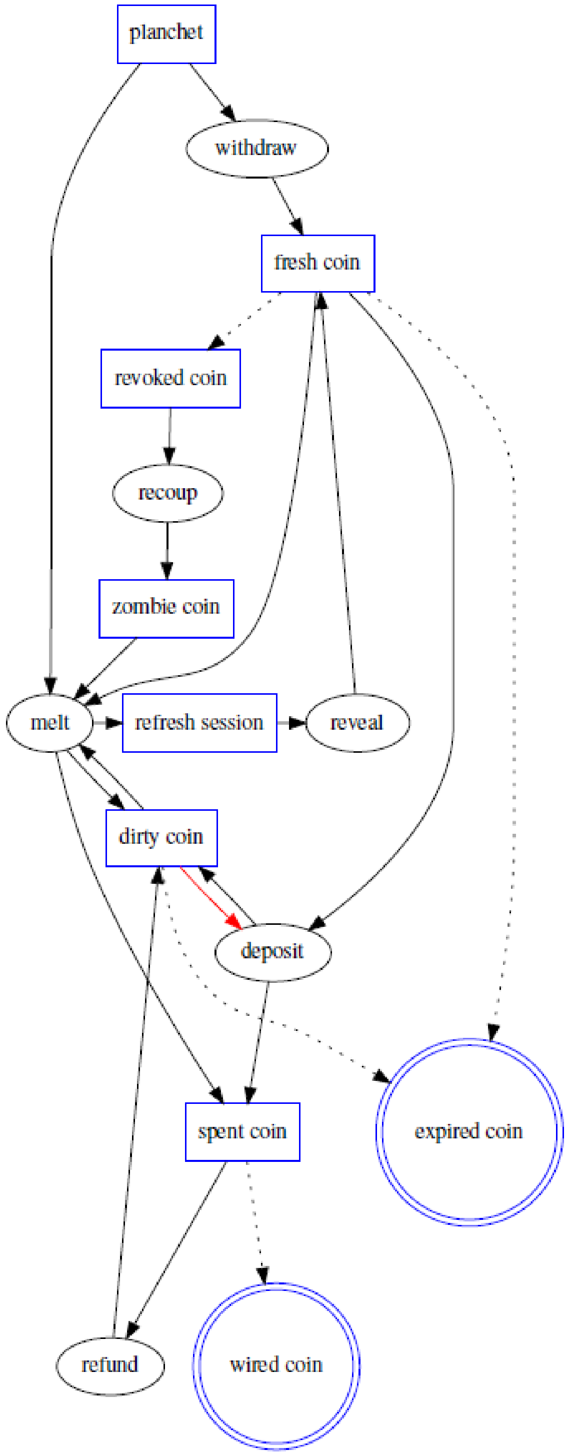

Figure 2.4.: Overview of states and state changes of coins [13]

The lifecycle of a coin starts with a planchet, which is created by a wallet if it wants to withdraw funds from the exchange. The exchange then signs the planchet, creating a fresh coin.

Once a coin is created several things may happen to it. It can, for example, expire, if it is not used within a certain time period. It may be refreshed by an exchange.

A coin, or rather its signature, may be revoked. The customer has the ability to recoup this coin and get a zombie coin, which can then be melted. If such a coin is spent, it can also be melted directly.

Finally a coin can be spent. By depositing it, it becomes a dirty or a spent coin. The coin is considered dirty if the public key is shared in some way, and spent if it is not.

Lastly, a spent coin can, through the refund protocol, become a dirty coin or a wired coin. Which like the expired coin state, is one of the two possible end states for a coin and this means it's life cycle is complete.

2.5.3. Deposit

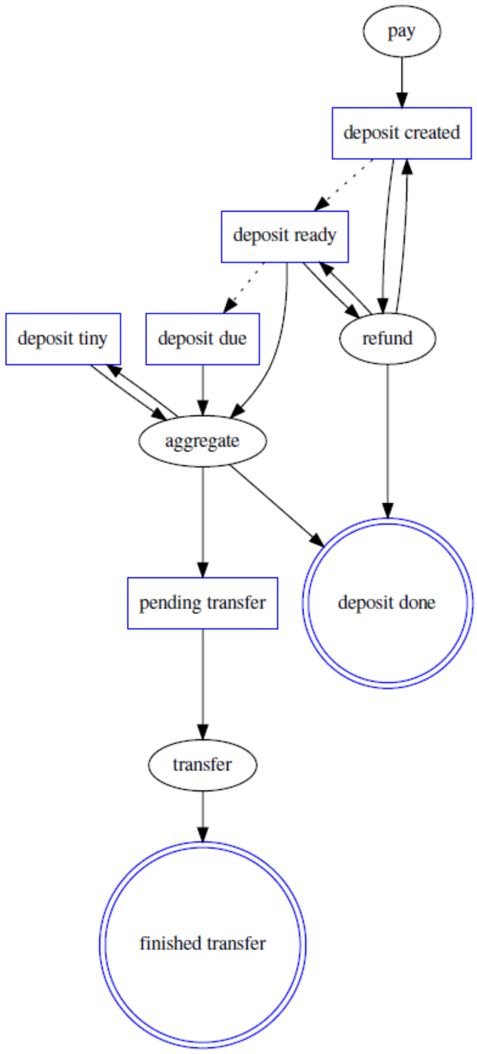

Figure 2.5.: Overview of states and state changes of deposits [13]

The depositing process is initiated with a payment. As soon as this process completes, a deposit is successful. In the 'ready' and 'due' state, it can aggregate and thus reach the 'tiny', 'done' or 'pending transfer' state. Once the transfer is finished, the deposit is complete.

A not yet completed deposit may also go through the refund process, which may or may not be successful, or reach the 'deposit done' state.

2.6. Description of Helpers

The work of the auditor takes places in its six helper programs, namely the helper deposits, coins, aggregation, wire, purse and reserves. Each helper has its own responsibilities and tests it performs, to find potential manipulation or misbehaviour.

Each helper roughly perform the same steps. First, they check their current auditing progress, as to not do the same work twice. Then tests are performed, and lastly auditing results are stored and their progress is updated. The helpers also piece together their own version of some of the exchanges database, like the exchanges current balances, reserves and so on, these are also updated as tests are being performed.

2.6.1. Helper Aggregation

The helper aggregation audits the exchanges aggregation activity. It includes the following test cases:

Check that a wire transfer made by the exchange is valid

This test first checks if a wire transfer has a transfer method, then compares the payto method with the payto URI. If they don't match, the auditor reports a 'row inconsistency'. Afterwards the auditor tries to find details, the denomination key and history for said coin claimed in the aggregation. If it cant find them, a 'row inconsistency' is reported here as well. The test goes on to lookup the technical details of the coin, trying to find wrong denomination keys, expired coins or bad signatures, which will result in a 'bad-sig-losses' report.

If it finds an invalid coin denomination signature, it will report a 'row inconsistency' again. Afterwards, it compares the coin and its paid fee, with the actual deposit fee, to find and report a 'amount arithmetic inconsistency' if they do not match.

Then, the helper checks other details of the wire transfer, like comparing the outgoing wire transfer target with the hash of the wire from the deposit while also comparing given wire transfer dates. If dates do not match a 'row inconsistency' is reported. The last check does a comparison of the given and the calculated amounts, which in turn can lead to a 'wire out inconsistency', if they differ.

Lookup the wire fee that the exchange charges at a timestamp

To validate wire fees, they are looked up in the exchange, if this is not possible for some reason, this leads to a 'row inconsistency' report. If an invalid or negative fee is reported back after subtracting the fee from the sum of all transactions by the given wire transfer id, an 'amount arithmetic inconsistency' report is generated.

Then, signatures of a wire fee at a given time are checked, if any of them fail, a 'row inconsistency' is reported. Next, the helper compares the given fee start and end dates, which can result in a 'fee time inconsistency' report if they dont make sense. This can happen if either the start date is earlier than the previous end date, or the end date is later than the next start date.

Check coin transaction history for plausibility

To check the coin transaction history, this test iterates over all given transactions and then computes the deposit and melt values, as well as the refund values.

A 'row inconsistency' is reported, when multiple instances of the same coin are detected in the same deposit. An 'amount arithmetic inconsistency' is reported if there is a disagreement in the given fee structure and the computed one, either in the deposit, melt or refund values.

This test also checks if the difference between refund values and deposit values is zero, if it is not, this leads to a 'coin arithmetic inconsistency'. Following up on these calculations of total balances, the last checks are a comparison of refunds and expenditures. A 'coin arithmetic inconsistency' is reported in case they don't match.

2.6.2. Helper Coins

This helper checks for all coin use cases. Signatures, denominations, blind signature tests etc.

Check withdrawal operations

This check examines, whether the coin's denomination key is missing. A 'row inconsistency' is reported if so.

Audit refund's execution

It inspects if the denomination key is missing, a 'row inconsistency' is reported if so. Then, the refund signature is verified, which may lead to a 'bad sig loss' report. An 'amount arithmetic inconsistency' is reported, if the amount without fee, subtracted with the amount with fee, does not correlate with the given refund fee. Further, if the denomination key for the refunded key is not known to the auditor, a 'row inconsistency' is reported.

Audit purse refund's execution

If the denomination key is missing, a 'row inconsistency' is reported. If it is unknown to the auditor, a 'row inconsistency' is reported.

Audit about recoups of refreshed coins

Is the denomination key of the old coin missing, a 'row inconsistency' is reported. After this, the coin's signature is verified, if the verification fails, it leads to 'bad sig loss' report. Then the recoup signature is verified, potentially resulting in a 'row inconsistency' report. If a coin is invalid – meaning the denomination key either doesn't exist, is expired or the signature is incorrect – a 'bad sig loss' is reported. Next, if the denomination key for recouped coin is unknown to auditor, 'row incosistency' is reported. The last check tests if there was a revocation of a signature that was not forwarded to the denomination, this would then lead to a 'bad sig loss' report.

Check the refresh execution

It starts with trying to find the denomination key, is it missing, it generates a 'row inconsistency'. Is the melting signature incorrect, a 'bad sig loss' is reported.

If the melting fee is higher than the contribution of the melted coin, an 'amount arithmetic inconsistency' is reported. If the refresh cost was higher than the amount without fee and the exchange made a loss, another 'amount arithmetic inconsistency' is reported. Next, the test checks again if the denomination key for the fresh coin is unknown to auditor, or the denomination of the dirty coin is unknown to it; a 'row inconsistency' report is generated.

Audit deposit execution

The test attempts to find the denomination key, which may result in a 'row inconsistency' report if it doesn't. The same report will also be generated, if the refund deadline is past the wire deadline. A 'bad sig loss' is reported when the deposit signature is invalid.

Audit purse deposit execution

Again, the check for the denomination key runs first and the signature check second. If the denomination key for a purse-deposited coin is unknown to the auditor after updating the denomination balance, a 'row inconsistency' report is generated.

Check the coin and its history

First the coin's history is calculated. Then, in case we detected a loss for the coin, an 'amount arithmetic inconsistency' report is generated.

2.6.3. Helper Deposits

The helper deposit is the simplest of all helper programs. It has one test case only:

Check that the deposit confirmation exists in the exchanges database

This test queries the deposit confirmations that were provided to it by merchants and checks that for each coin used in that deposit, it can find the same transaction in the exchanges database. If there is one missing, it leads to a reported 'deposit confirmation inconsistency'.

2.6.4. Helper Wire

The helper wire audits the reserve's closing operations triggered by the aggregator. Those run through some tests, while the helper gets its data not only from the replicated exchange database, it also gets the data from the bank API. It goes over all bank accounts and checks for deltas and other indicators. As the helper wire is structured a bit differently than the other helpers, it's more understandable to display it's tests in a list:

- A 'closure lag' is detected and reported, if there were any entries found in reserves closures, that were not yet observed.
- A 'KYC lag' is reported, if there is a kyc entry in the wire transfers that should have been performed.
- An 'AML lag' is reported, if there is an aml entry in the wire transfers that should have been performed.
- A 'lag' is reported, if a lag is detected in the wire transfer.
- A 'row minor inconsistency' is reported, if any kind of timing anomalies were detected.
- A 'wire out inconsistency' is reported, if any outgoing wire transfer was not yet made, but could or should have been.
- A 'wire out inconsistency' is reported, when there is a receiver account mismatch found on both sides.
- A 'wire out inconsistency' is reported, when the wire amounts do not match.

- A ’row inconsistency’ is reported, if there was a profit drain found. Meaning a wire transfer happened, that was not allowed to, because a signature was missing or invalid.
- A ’wire out inconsistency’ is reported, if a transfer was found with a delta in target accounts.
- A ’wire out inconsistency’ is reported, if a profit drain with an incorrect amount was found.
- A ’wire out inconsistency’ is reported, if the jurisdication of a wire transfer was not found.
- A ’wire format inconsistency’ is reported, if there was a format error of a wire transfer.
- A ’row inconsistency’ is reported, if a duplicated wire offset was found.
- A ’reserve in inconsistency’ is reported, when an incoming wire transfer claimed by the exchange was not found.
- A ’reserve in inconsistency’ is reported, if there is a delta in wire transfer subjects, on both sides.
- A ’reserve in inconsistency’ is reported, if there is a delta in the wire amount.
- A ’misattribution in inconsistency’ is reported, if there was a misattribution found.
- A ’row minor inconsistency’ is reported, when the execution dates do not match.
- A ’row minor inconsistency’ is reported, if the given closing fee is above the total amount.

2.6.5. Helper Purses

In this helper, purses are checked.

Handling of a purse’s requests

Verifyies a purses the signatures. If they are invalid, a ’bad signature loss’ report is generated.

Audit a purse’s merge execution

Tries to verify each purse merge by recomputing it and comparing the signatures. If they are invalid, a ’bad signature loss’ report is generated. Finally, the auditor tries to create a new reserve for the given reserve public key. If it fails, it reports a 'row inconsistency'.

Audit an account's merge execution

Audits account merges and tries to verify its signatures and on failing, a 'bad signature loss' report is generated.

Audit a purse's decision

With all purse refunds loaded from the database, the test first tries to setup the purse, possibly resulting in a 'row inconsistency' report. Then, the purse fee for the purse created at the given time will be queried, to check if the fee is available or not, which if not, results in a 'row inconsistency' report. If the fee is available but higher than the balance, another 'row inconsistency' is reported. The last two checks compare the values of a purse, either the refund or the merge values, if they don't match, this results in an 'amount arithmetic inconsistency'.

Audit expired purses

An expired purse, that was not closed, immediately leads to a 'purse not closed inconsistency' report.

Purse balance summary check

Finally the last purse check does an iteration over all purses and checks if it can query their respecitve fees and if not, this results in a 'row inconsistency'. It goes on to subtract the fee from the balance to get the actual balance it expects and tests, if the purse fee is higher than the given balance. If so, a 'row inconsistency' report is stored. The last check compares the purse's exchange balance amount with the balance amount given without the fee, if they don't match up, an 'amount arithmetic inconsistency' is reported.

2.6.6. Helper reserves

The helper reserves audits reserves for being well-formed.

Audit withdrawals

The test starts by checking for the denomination key, if it is not found, a ’row inconsistency’ is reported. It goes on to check the execution date of a withdrawal, if it is not within the allowed range, it leads to a ’denomination key validity withdraw inconsistency’.

Audit recoup operations by reserve

First, the coin’s signature is verified, looking for a ’bad sig loss’. Second, a ’row inconsistency’ is reported if the revocation set does not include the denomination key. Third, another ’bad sig loss’ is reported, if the master signature is invalid.

Test reserve opening operations

If the reserves operation specific signature is invalid, a ’bad sig loss’ is reported.

Test reserve closing operations

The fee of the reserve closing operation is checked for deltas in given and expected values, potentially resulting in an ’amount arithmetic inconsistency’ report. While the reserve closing request is unknown to the auditor, a ’row inconsistency’ is reported. Another ’bad sig loss’ is reported, if the signature of the closing request is invalid. Lastly, the test reports ’row inconsistencies’ for the following cases: the target account is not verified and auditor does not know the reserve, or the target account does not match its origin account in sender and receiver.

Checks account merge requests

It checks the reserve’s signature, which leads to a ’bad sig loss’ if the verification fails.

Verify reserve balances

A ’reserve balance insufficient inconsistency’ is reported, in case of given balances not matching, either in negative or positive.

A ’reserve balance summary wrong inconsistency’ is noticed and reported, if the computed and given amounts do not match.

A ’reserve not closed inconsistency’ is found and reported, when either the remaining reserve balance exceeds the closing fee, or the closing fee could not be determined.

3. Solution Design

3.1. Architecture

A lot of changes to the existing auditor architecture were necessary, since the auditor should now write into a database instead of JSON files.

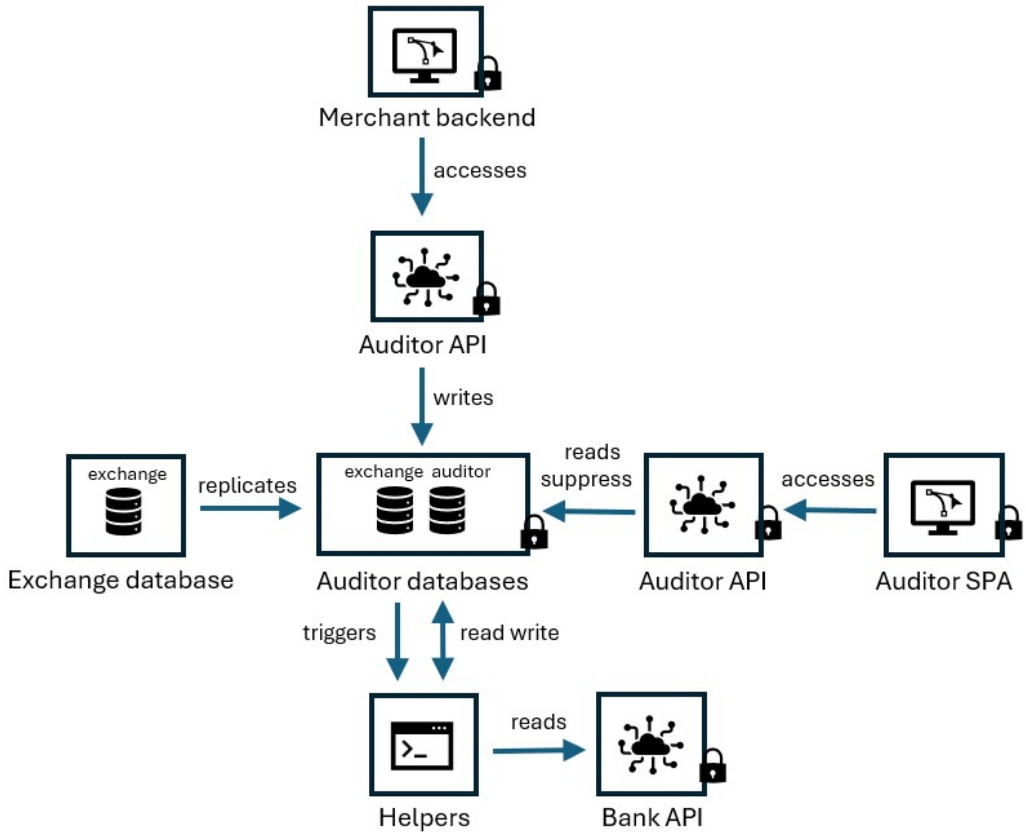

Figure 3.1.: New Auditor architecture

The new architecture is dictated by the exchange's database. The new program flow is to be understood like this:

1. On an insert into the exchange database, the helpers get triggered to do their work
2. After auditing, they write back their findings to the auditor database
3. Through the API, the SPA reads from the database constantly and presents the auditors findings in it's dashboard

Every database, API and SPA access is protected by a bearer token. This provides basic security, enough that the data is protected inside the operater's network. The idea is, that the auditor is run behind a reverse proxy anyway, which means, that the access control is managed at this front and not needed at the auditor's side.

3.2. Auditor database

The auditor's database schema is based on the current behaviour of the helpers. They generated JSON reports with different sections, these sections are now database tables. The attributes of the tables reflect the data in the code.

For the database table's insert triggers and the required event subscription in the code, we followed the official PostgreSQL documentation and used the existing GNUnet event subscription code base.

3.3. REST API

The procedure in designing was clear. First, start by documenting the needed endpoints and afterwards, extend the auditor's webserver codebase and at last, update and extend the REST API. The needed endpoints were based on the incident categories. Only for incidents and balances, a PATCH operation was provided. A bearer token for security will be added, for those requests that are not designed to be publicly accessible. The format and responses from endpoints were designed to adapt to existing GNU Taler APIs.

3.4. SPA

Since GNU Taler has different components with SPA's already, the idea was to align the codebase and technologies, as well as it's design, to improve maintainability and recognition. So it's code was partially taken from the GNU Taler merchant [14] and adjusted to the auditor's needs.

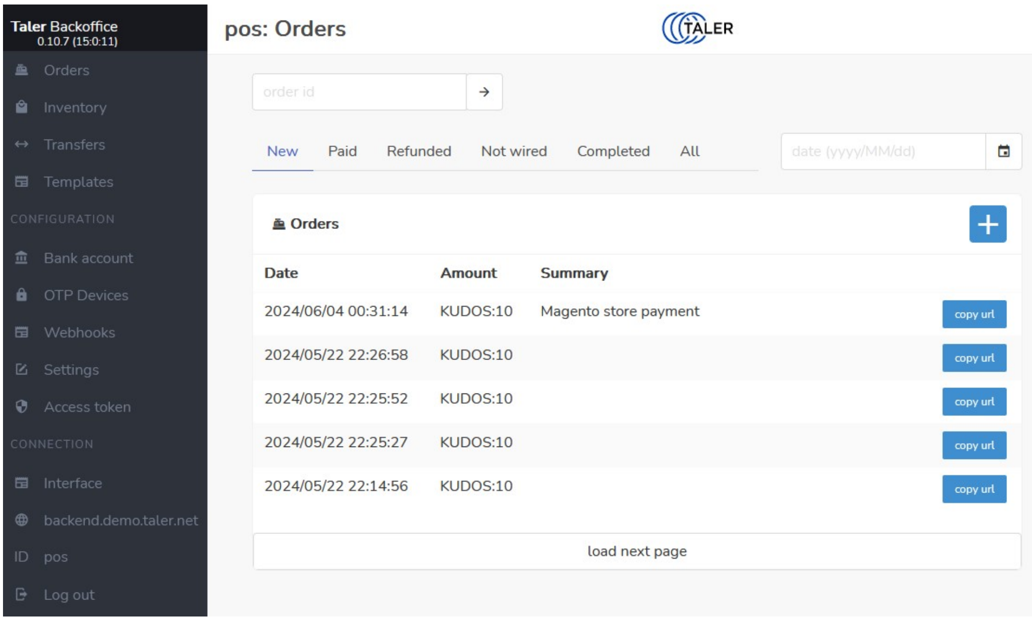

Figure 3.2.: Merchant SPA

The main point was not the design, but the data to be presented. While the design stayed the same, with the navigation on the left, the header on top and the content in the middle, as well with the same looks, the real task was to connect that design with the auditor's data and present it in the best possible way, for the data to be understood.

3.4.1. Data to Display

Data is divided into four categories, each representing a dashboard that is navigatable like in figure 3.2.

The four dashboards are:

1. Key figures, for the management and analysts, interested in tracking the exchanges gains and losses
2. Critical incidents, where business impacting incidents are shown, tracked and investigated.
3. Monitoring, exploring the protocol and network state finding difficulties and operating problems
4. Detail state, to go in depth

4. Implementation

4.1. Overview

The real-time auditor stores results of it's audits in PostgreSQL tables, every inconsistency that the auditor looks for, has a designated table. These, along with any other databases are set up when the auditor is started. Helpers are written in C, and thus communicate with the PostgreSQL database via an interface based on the libpq C library. All Helpers, except the deposit helper, only add or update elements in the database or get them from it. The deposit helper can also delete elements from it's tables.

To see results of the real-time auditor, a webpage continuously fetches elements form the auditor database. A small microhttp server handles requests to the database.

Requests from the web are only allowed to GET elements or PATCH a specific field that indicates whether an element should be sent again on subsequent GET requests or not.

4.2. Implementation of tables

4.2.1. Overview

There are more than 20 tables the GNU Taler auditor writes to. That does, however, **not** equate the number of issues the auditor actually tracks.

The different tables do give an idea of what errors are recognized, but there are also some minor issues that are not seperately categorized, and instead collected in general tables. Other tables store no errors at all, but instead contain information about the internal state of the auditor itself. And lastly, some contain records about balances or reserves etc, which the auditor then compares to the exchange's records.

As a result of this, it is important to recognize that, when the auditor adds a new row to one of it's tables, it does not automatically mean some crime has been committed, or fraud has taken place. This is why any critical developments are surfaced in a single page application, for a human to review.

In the following chapters, an overview and the structure is given over the newly created tables the auditor writes to and what an entry in each of them means.

Some of the descriptions have been taken from the exchange's documentation itself, while others have been provided or extended by Prof. Dr. Christian Grothoff. These descriptions are also available seperately in the documentation of the GNU Taler auditor REST API.

4.2.2. Monitoring Status

Arithmetic Inconsistencies

This table contains cases where the arithmetic of the exchange involving amounts disagrees with the arithmetic of the auditor. Disagreements imply that either the exchange made a loss (sending out too much money), or screwed a customer (and thus at least needs to fix the financial damage done to the customer). The profitable column is set to true if the arithmetic problem was be determined to be profitable for the exchange, false if the problem resulted in a net loss for the exchange.

Losses Caused by Invalid Signatures

This table contains operations that the exchange performed, but for which the signatures provided are invalid. Hence the operations are invalid and the amount involved could be a loss for the exchange (as the involved parties could successfully dispute the resulting transactions).

Closure Lags

A closure lag happens if a reserve should have closed a reserve and wired (remaining) funds back to the originating account, but did not do so on time. Significant lag may be indicative of fraud, while moderate lag is indicative that the systems may be too slow to handle the load. Small amounts of lag can occur in normal operation.

If closure lag is experienced, the administrator should check that the **taler-exchange-closer** component is operating correctly.

Coin Inconsistencies

This table contains cases where the exchange made arithmetic errors found when looking at the transaction history of a coin. The totals sum up the differences in amounts that matter for profit/loss calculations of the exchange. When an exchange merely shifted money from customers to merchants (or vice versa) without any effects on its own balance, those entries are excluded from the total.

Denomination Key Validity Withdrawal Inconsistencies

This table highlights cases, where denomination keys were used to sign coins withdrawn from a reserve before the denomination was valid or after it was already expired for signing. This doesn't exactly imply any financial loss for anyone, it is mostly weird and may have affected the fees the customer paid.

Denominations Without Signatures

This table highlights denomination keys that lack a proper signature from the taler-auditor-offline tool. This may be legitimate, say in case where the auditor's involvement in the exchange business is ending and a new auditor is responsible for future denominations. So this must be read with a keen eye on the business situation.

Deposit Confirmations

This table contains a list of deposits confirmations that an exchange provided to merchants but failed to store in its own database. This is indicative of potential fraud by the exchange operator, as the exchange should only issue deposit confirmations after storing the respective deposit records in its database. Not storing the deposit data means that the exchange would not pay the merchant (pocketing the money) or allow the customer to double-spend the money (which is naturally also not good).

Note that entries could appear in this list also because the exchange database replication is delayed. Hence, entries that are only a few seconds old might not be indicative of an actual problem. If entries in this list are more than a few seconds old, the first thing to check is whether or not the database replication from the exchange is working properly.

Incoming Misattributions Inconsistencies

This table contains cases where the sender account record of an incoming wire transfer differs between the exchange and the bank. This may cause funds to be sent to the wrong account should the reserve be closed with a remaining balance, as that balance would be credited to the original account.

Purse not Closed Inconsistencies

This table highlights cases, in which either payer or payee did not finish their part of a P2P payment. This caused a purse –– which may contain some money — to reach its expiration date. However, the exchange failed to properly expire the purse, which means the payer did not get their money back. The cause is usually that the **taler-exchange-expire** helper is not running properly.

Refreshes Hanging

This table highlights cases, where a coin was melted but the reveal process was not finished by the wallet. Usually, a wallet will do both requests in rapid succession to refresh a coin. This might happen, even if the exchange is operating correctly, if a wallet goes offline after melting. However, after some time wallets should in most cases come back online and finish the operation. If many operations are hanging, this might be indicative of a bug (exchange failing on reveal, or wallets not implementing refresh correctly).

Reserve Balance Insufficient Inconsistencies

This table highlights cases where more coins were withdrawn from a reserve than the reserve contained funding for. This is a serious compromise resulting in proportional financial losses to the exchange.

Reserve Balance Summary Wrong Inconsistencies

This table highlights cases, where the exchange's and auditors' expectation of the amount of money in a reserve differs.

Reserve in Inconsistencies

This table contains cases where the exchange's and auditor's expectation of amounts transferred into a reserve differs. Basically, the exchange database states that a certain reserve was credited for a certain amount via a wire transfer, but the auditor disagrees about this basic fact. This may result in either a customer loosing funds (by being issued less digital cash than they should be) or the exchange loosing funds (by issuing a customer more digital cash than they should be).

Reserve not Closed Inconsistencies

This table highlights cases, in which reserves were not closed, despite being expired. As a result, customers that wired funds to the exchange and then failed to withdraw them are not getting their money back. The cause is usually that the **taler-exchange-closer** process is not running properly.

Row Inconsistencies

This table highlights inconsistencies in a specific row of a specific table of the exchange. Row inconsistencies are reported from different sources, and largely point to some kind of data corruption (or bug). Nothing is implied about the seriousness of the inconsistency. Most inconsistencies are detected if some signature fails to validate. The affected table is noted in the 'table' field. A description of the nature of the inconsistency is noted in 'diagnostic'.

Minor Row Inconsistencies

The section highlights inconsistencies where a row in an exchange table has a value that is does not satisfy expectations (such as a malformed signature). These are cause for concern, but not necessarily point to a monetary loss (yet).

Wire Format Inconsistencies

This table highlights cases where the wire transfer subject was used more than once and is thus not unique. This indicates a problem with the bank's implementation of the revenue API, as the bank is supposed to warrant uniqueness of wire transfer subjects exposed via the revenue API (and bounce non-unique transfers).

Wire Out Inconsistencies

This table highlights cases where the exchange wired a different amount to a destimation account than the auditor expected.

4.2.3. Critical Errors

Emergencies

Emergencies are errors where the total value of coins deposited (of a particular denomination) exceeds the total value that the exchange remembers issuing. This usually means that the private keys of the exchange were compromised (stolen or factored) and subsequently used to sign coins off the books. If this happens, all coins of the respective denomination that the exchange has redeemed so far may have been created by the attacker, and the exchange would have to refund all of the outstanding coins from ordinary users. Thus, the risk exposure is the amount of coins in circulation for a particular denomination and the maximum loss for the exchange from this type of compromise.

The difference between emergencies and emergencies by count is how the auditor detected the problem: by comparing amounts, or by counting coins. Theroretically, counting coins should always detect an issue first, but given the importance of emergencies, the auditor checks both total amounts and total numbers of coins (they may differ as coins may be partially deposited).

Emergencies By Count

Emergencies "by count" are cases where this type of money printing was detected simply by counting the number of coins the exchange officially put into circulation and comparing it to the number of coins that were redeemed. If the number of redeemed coins is higher than the number of issued coins, the auditor reports an emergency-by-count.

Fee Time Inconsistencies

This table highlights cases where validity periods associated with wire fees the exchange may charge merchants are invalid. This usually means that the validity periods given for the same type of fee are overlapping and it is thus unclear which fee really applies. This is a sign of a serious misconfiguration or data corruption as usually the exchange logic should prevent such a fee configuration from being accepted.

4.2.4. Operational Status

Balances

Returns the various balances the auditor tracks for the exchange, such as coins in circulation, fees earned, losses experienced, etc.

Historic Denomination Revenue

This endpoint is used to obtain a list of historic denomination revenue, that is the profits and losses an exchange has made from coins of a particular denomination where the denomination is past its (deposit) expiration and thus all values are final.

Historic Reserve Summary

This endpoint highlights cases, where the exchanges expectation of the summary in a reserve differs from its actual summary.

Progress

This endpoint contains information about the auditing progress an auditor has made.

Reserves

This endpoint is used to obtain a list of reserves.

Purses

This endpoint is used to obtain information about open purses.

Pending Denominations

This endpoint is used to obtain a list of balances for denominations that are still active, that is coins may still be deposited (or possibly even withdrawn) and thus the amounts given are not final.

4.3. Interfaces

The old auditor would store findings in memory, until it saved them to a JSON file, whereas the real-time version saves any findings in dedicated PostgreSQL tables as soon as they are discovered. Then, the contents of the tables are displayed in a webportal, that continuously updates.

The connection between the tables, the auditor and the webportal is facilitated through two key interfaces; A REST API and a PostgreSQL C API.

A REST API allows the webportal to request table entries in JSON format from the auditor, so that it can then display them. The webserver that receives those requests must fetch elements from a database, and it does so with a PostgreSQL C API.

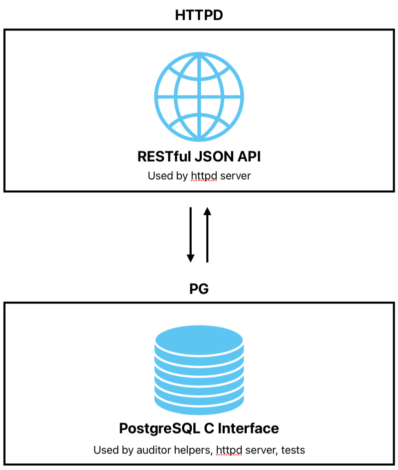

Figure 4.1.: Interaction Between Auditor Components

4.3.1. REST API

Only GET and PATCH functionality is strictly required by the webportal, which is described in more detail in chapter 4.5. For testing purposes, PUT and DELETE functions were also added, and subsequently disabled.

GET

All GET requests added as part of this project have the same structure. As an example, with the endpoint http://localhost:8083/monitoring/emergency (provided the auditor runs on the local machine of course) one receives at most 20 items, starting with the newest, from the emergency table. The same logic applies to all other inconsistencies – or tables – the auditor records.

Three query arguments, can be used to customize a response:

limit A signed integer. Specifies how many elements should be returned, relative to the offset argument. The default is -20.

offset An unsigned integer. Specifies from which row onwards to return elements. The default is INT_MAX, meaning the latest element.

return_suppressed A boolean. If true, then all elements are returned, regardless of whether or not they were suppressed. The default is false.

Figure 4.2. demonstrates how the parameters limit and offset can be used together to retrieve any contiguous number of rows. In the example, the offset is 40. If limit is chosen to be a negative number, like -20, the rows with row_ids 20 to 40 would be returned. A positive limit of ten would return rows 40 to 50.

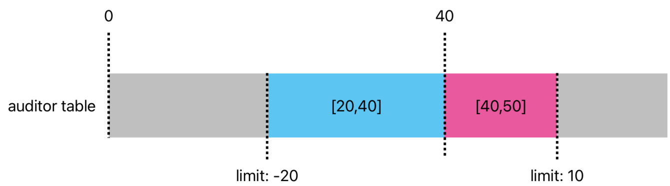

Figure 4.2.: Using offset and limit query arguments

The 'bad-sig-losses' table required some additional customisation. Two the additional query arguments 'op' of type string and 'use_op_spec_pub' of type boolean were added to the GET request. If 'use_op_spec_pub' is sent as an argument, then an operation specific public key encoded with Crockford's version of Base32 Crockford Base32 must be given in the requests' body.

With these arguments, the returned objects can be restricted to include only those that contain a certain operation string ('op') or public key associated with an operation ('use_op_spec_pub'). Both of these additional query arguments are optional.

The balances GET request and database query, required the addition of a 'balance_key' query argument. If this optional query argument is specified, only balances containing this key are returned.

PATCH

As the auditor runs, some tables might accumulate many rows. To only show rows that have not been seen yet, it is possible to 'supress' old entries from the webpage. A row that is suppressed is not shown again in the future unless specifically requested.

This is done with a PATCH request, with which an entry of a table of the auditor can be altered in a predetermined way. The only fields that can be changed with this request are the 'suppressed' fields of a table. As an example, in the table emergency by count, to change the second rows' suppressed value to true, one would call the following endpoint: http://localhost:8083/monitoring/emergency-by-count/2 (again, assuming the auditor runs locally)

In the body of the request, one can send a very simple JSON object that looks like this:

```
{
    "suppressed" : true
}
```

One could also unsuppress a row, by setting the value to false.

Not every endpoint can be suppressed. Chapter 4.5 further elaborates, how endpoints are divided into groups. Only entries in tables that store actual emergencies or errors can be suppressed. It makes no sense to suppress internal consistency information the auditor stores for itself.

4.3.2. PostgreSQL C API

The PostgreSQL C API enables interaction between C and the auditors PostgreSQL tables. This API is used by the webserver of the auditor as well as the helpers and tests.

It exposes at most four functions for each table, one to get rows from the database, one to add rows to it, one to update a row and one to delete rows. Not all tables support all these functionalities.

Select

The query parameters from the the REST GET requests can be used here, to retrieve the correct elements with a SELECT statement. A JSON object is returned.

Insert

Allows one to insert an element into the PostgreSQL database via a database query. Values like row_id and suppressed (where applicable) are automatically generated by PostgreSQL, and must not be inserted.

Update

For most tables, this function is closely related to the PATCH function in the REST API. The only thing this function updates is the 'suppressed' field of any table of the auditor. Though, some tables do support updating other fields as well. This way, an entry can be updated with new values, instead of creating a new one.

Delete

With this function, it's possible to delete one row at a time from a given table. Right now, this function is not actually used by any helpers, except for the deposit helper, which deletes rows it already processed from it's database.

4.4. TRIGGERS, LISTEN and NOTIFY

At the heart of the real-time logic are PostgreSQL triggers, that fire if new data is added to certain tables of the **exchange**.

Under normal operation, helpers are dormant, but listen to specific triggers through event handlers. If a PostgreSQL trigger activates, these event handlers are called, and the helper begins its analysis. Some tables in the exchange's database trigger more than one helper to wake up.

4.5. Single Page Application

4.5.1. Description

The auditor continuously monitors changes in the exchange database, and writes any suspicious behaviour in its database. A small website was built, to display these results in an easily digestible way.

4.5.2. Technologies

Within GNU Taler, some systems already use single page applications, meaning templates could be used to make this single page application similar to other components' frontends. As a result we used Preact, TypeScript and Scss. As for the used server technologies, node.js and the Taler internal webserver, which is based on microhttp, were used.

4.5.3. Implementation

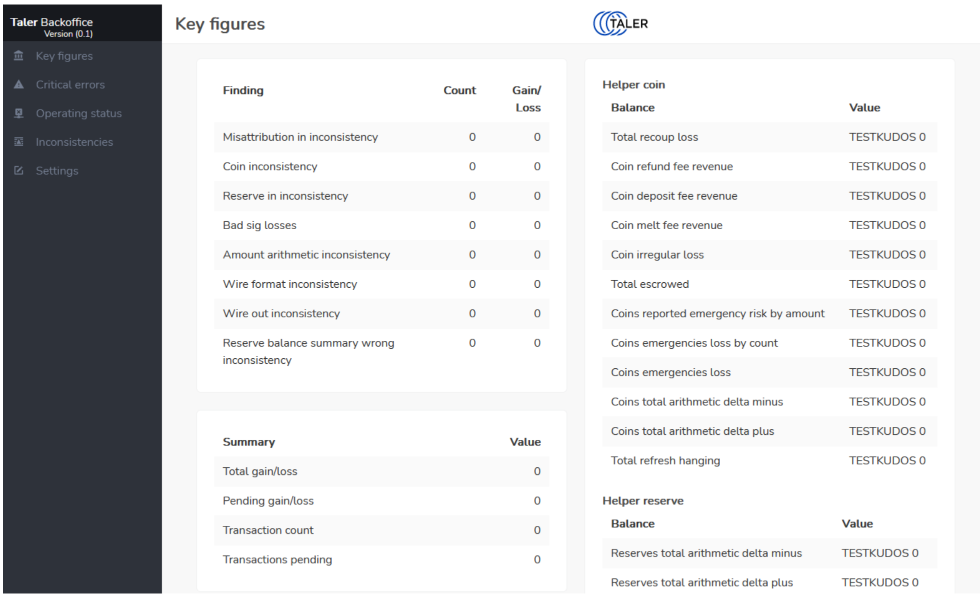

Figure 4.3.: Dashboard key figures

Data from the auditor is divided into several categories. Key figures displays general info about the exchange, the critical errors and inconsistencies tabs show suspicious things the auditor detected in it's audits. Operating status shows status information about the auditor itself. Because all tables in the auditor database have different columns, they do not always display the same information, even if they are in the same category.

4.5.4. Authentication

Figure 4.4.: Bearer token implementation

Users of the webportal can add a bearer token via a textfield, so the auditor API can be accessed. When first launching the site, a popup asks for the token and validates it, before granting access to the application. The implementation can be seen in figure 4.4..

4.5.5. Dashboards

The implementation of the dashboards per group was organized per their data. The focus was on showing the most important values directly, but still displaying the data in full and allowing for a complete analysis. In the key figures dashboard, we see all findings with their count and their gains or losses (see figure 4.3.).

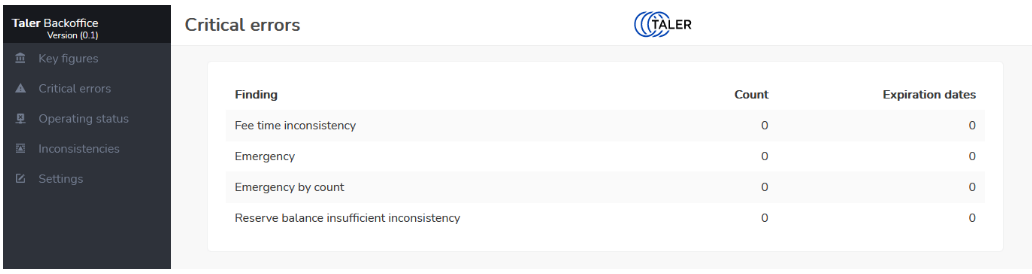

Figure 4.5.: Dashboard critical error

For the critical errors, the focus lay on presenting the worst possible errors.

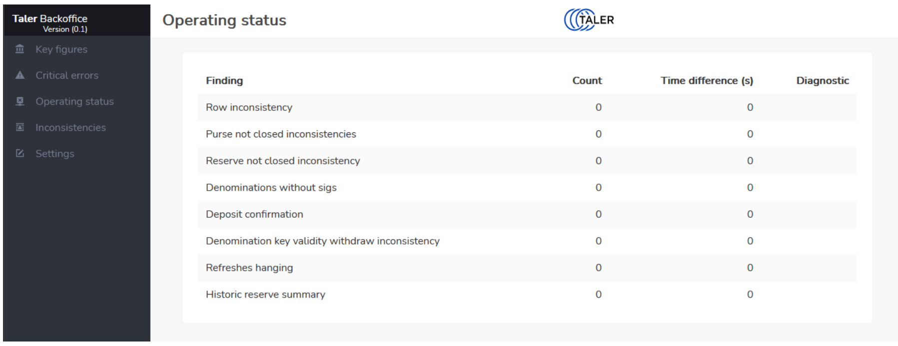

Figure 4.6.: Dashboard operating status

The operation's view dashboard shall give a quick overview over the state of the network and it's operating status. Thus it displays the counts of operating status findings, their potential time difference and diagnostic strings.

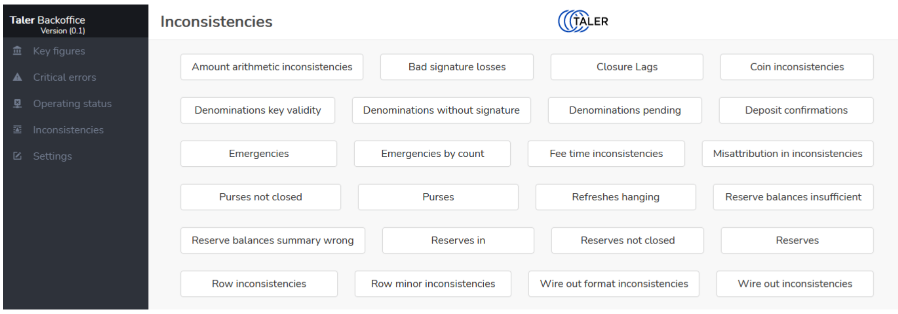

Figure 4.7.: Dashboard inconsistencies

Here, all possible auditor findings are displayed and can be investigated further, leading to a full view of each table status.

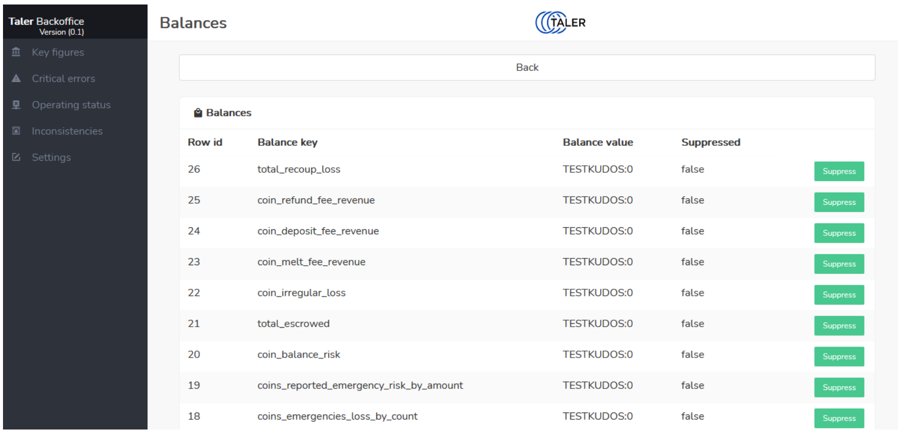

Figure 4.8.: Finding detail view

5. Discussion

5.1. Approach

Adding these tables and functions amounted to so many new files and additional pieces of code across many existing ones, that a python script was used to generate some of the required C code. This was especially easy for the PATCH and DELETE HTTP functions, since they needed no customization, except for the name of the table they affected. Adapting the script to produce code for the GET and PUT functions was more difficult, and still required some manual intervention afterwards.

To actually generate code, the scripts read from the sql files that contained the auditor tables, extracted information like column names and types or table names and filled those into string templates. However, because C structures like hashes, EdDSA [15] signatures or EdDSA keys are all stored as byte arrays in PostgreSQL, the scripts could not infer those types when generating C code that required them. This had to be corrected manually.

The documentation of the REST JSON API of the auditor was also generated with the help of a python script. It too, worked by extracting relevant table columns, types and names from SQL files and inserting them into a string template. Though significant changes and additions were necessary in the documentation as well.

5.2. Future Work

Despite the progress made in this project, there are also a lot of things that could be addressed in future projects.

The webportal, for example, could display more detailed information still, and perhaps enjoy some usability upgrades. Also, the webpage could, instead of periodically polling the auditor database, receive notifications from the HTTP server if new data is available, and then fetch it when needed. Another useful feature the auditor could provide, is using push notifications or emails to alert exchange operators as soon as emergencies are detected. Also, a proper dataset could be set up, to further fine tune the frontend by analyzing the data and finding further insights. A big difference could potentially make the extension of the auditor's data model by historical auditor data to show the development, usage and operating history of the exchange.

The tests to check if the helpers are working correctly could also be improved. Some existing tests are not working properly, and should be fixed. Perhaps more tests could be added to find more edge cases, or constellations which are not yet caught by existing ones. Like finding auditor idempotency cases and storing them.

Work could be done to parallelize the helpers' analysis, with the intention of making them faster. Either, parallelization could be done solely on the CPU, or some calculations could even be offloaded to the GPU and free up resources on the main processor. Though parallelization on the GPU might promise large performance gains, implementing the necessary features would not be trivial. Some of the helpers' responsibilities include verifying cryptographic signatures, which involves modular exponentiation with very large integer numbers. GPUs are not designed for such operations, and even though they might be able to verify many signatures at once, that advantage of parallelization might not be enough. Also, GPU programming is often generalized through frameworks like OpenGL / OpenCL [16]. This universal applicability comes with additional performance losses, compared to CPUs. Highly optimized algorithms developed specifically for a given GPU architecture, however, could perhaps yield acceptable results. This could be subject of a future paper.

6. Conclusion

This thesis not only showed the necessities a payment system auditor needs to have, but even more so, the state of existing payment methods and the limits of most modern technology implementations. This current auditor is now in a state where it can be used to test its production readiness and can be operated to audit instances of exchanges. Thus, we were able to add substantial improvements to the auditors capabilities and usability.

We believe that accountability is not just a commodity, but a necessity, especially when it comes to modern payment systems. We also believe that GNU Taler, and its auditor can deliver precisely these things. It seems however, that not everyone shares this simple notion with us.

Right now, the EU considers launching the Digital Euro [17], which is supposed to be a digital alternative to the Euro; in that sense, it would be much like GNU Taler. Crucially though, where the Digital Euro differs from GNU Taler, is the support for anonymous transitive **offline** payments. Such offline payments are virtually impossible to audit or conclusively verify, as a device that is offline may never be connected to the Internet, thus depriving auditors of the opportunity to inspect its state in a timely fashion. The problem is enhanced by the need to rely on hardware security modules with a horrible track record [18] as the CAP theorem by Eric Brewer, Seth Gilbert and Nancy Lynch [19, 20] makes it clear that maintaining consistency merely via software and protocols is impossible in this setting.

We can confidently say, that GNU Taler is the payment system we want to use and want to be used by society, going forward in the era of digital money. Or in other words: we know of no current existing payment system that protects data privacy, ensures security and offers a state of the art wallet and that we can put our trust in, due its licensing model, apart from GNU Taler. The GNU Taler auditor is an important part of the answer why society can trust the system, and other digital currency solutions should be evaluated with this level of auditability in mind.

All this is to say, that we think GNU Taler could help solve some of the shortcomings of payment systems today, and that the auditing philosophy behind it plays a vital part in that.

Erklärung der Diplomandinnen und Diplomanden
Déclaration des diplômant-e-s

Selbständige Arbeit / *Travail autonome*

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

Par ma signature, je confirme avoir effectué ma présente thèse de bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.) et autres ressources qui m'ont fortement aidé-e dans mon travail sont intégralement mentionnées dans l'annexe de ma thèse. Tous les contenus non rédigés par mes soins sont dûment référencés avec indication précise de leur provenance.

Name/*Nom*, Vorname/*Prénom* Eigel, Nicola Sacha

Datum/*Date* 06.06.2024

Unterschrift/*Signature*

Dieses Formular ist dem Bericht zur Bachelor-Thesis beizulegen.
Ce formulaire doit être joint au rapport de la thèse de bachelor.

Erklärung der Diplomandinnen und Diplomanden
Déclaration des diplômant-e-s

Selbständige Arbeit / *Travail autonome*

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

Par ma signature, je confirme avoir effectué ma présente thèse de bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.) et autres ressources qui m'ont fortement aidé-e dans mon travail sont intégralement mentionnées dans l'annexe de ma thèse. Tous les contenus non rédigés par mes soins sont dûment référencés avec indication précise de leur provenance.

Name/*Nom*, Vorname/*Prénom* Zwahlen, Cedric Vincent

Datum/*Date* 06.06.2024

Unterschrift/*Signature* C.Zw

Dieses Formular ist dem Bericht zur Bachelor-Thesis beizulegen.
Ce formulaire doit être joint au rapport de la thèse de bachelor.

Bibliography

[1] Bernie Madoff: Who He Was, How His Ponzi Scheme Worked — investopedia.com. https://www.investopedia.com/terms/b/bernard-madoff.asp. [Accessed 10-06-2024].

[2] Condé Nast. How the Biggest Fraud in German History Unravelled — newyorker.com. https://www.newyorker.com/magazine/2023/03/06/how-the-biggest-fraud-in-german-history-unravelled, 2023. [Accessed 10-06-2024].

[3] Sam Bankman-Fried and the FTX collapse, explained — nbcnews.com. https://www.nbcnews.com/tech/crypto/sam-bankman-fried-crypto-ftx-collapse-explained-rcna57582. [Accessed 10-06-2024].

[4] Derek Saul. First Republic Bank Failure: A Timeline Of What Led To The Second-Largest Bank Collapse In U.S. History — forbes.com. https://www.forbes.com/sites/dereksaul/2023/05/01/first-republic-bank-failure-a-timeline-of-what-led-to-the-second-largest-bank-c [Accessed 10-06-2024].

[5] GNU Taler team. Gnu taler website. https://taler.net/de/. Accessed: 2024-06-03.

[6] Florian Dold. The gnu taler system: practical and provably secure electronic payments. (le système gnu taler: Paiements électroniques pratiques et sécurisés). https://api.semanticscholar.org/CorpusID:195785269, 2019.

[7] Postgresql listen/notify. https://www.postgresql.org/docs/current/sql-notify.html. [Accessed 10-06-2024].

[8] Taler Systems team. Taler systems website. https://www.taler-systems.com/en/electronic-cash.html. Accessed: 2024-06-03.

[9] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[10] Wikipedia. Blind signature. https://en.wikipedia.org/wiki/Blind_signature. Accessed: 2024-06-03.

[11] GNU Taler team. Gnu taler developer manual. https://docs.taler.net/taler-developer-manual.html. Accessed: 2024-06-03.

[12] LibEuFin; GNU Taler. https://docs.taler.net/libeufin/index.html, 2014. [Accessed 10-06-2024].

[13] GNU Taler team. Gnu taler protocol descriptions. https://git.taler.net/exchange.git/tree/doc/system/taler. Accessed: 2024-06-06.

[14] GNU Taler team. Gnu taler merchant backoffice codebase. https://git.taler.net/wallet-core.git/tree/packages/merchant-backoffice-ui. Accessed: 2024-06-05.

[15] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Aug 2012.

[16] OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems — khronos.org. https://www.khronos.org/opencl/. [Accessed 12-06-2024].

[17] European Central Bank. Digital euro. https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html. [Accessed 10-06-2024].

[18] Researchers Discover Way to Hack Hardware Security Module, Gain Access to Cryptographic Keys. https://www.darkreading.com/identity-access-management-security/researchers-discover-way-to-hack-hardware-security-module-gain-access-to-crypto [Accessed 10-06-2024].

[19] E.A. Brewer. Towards robust distributed systems, folien zur keynote des 19. acm sigact-sigops symposium on principles of distributed computing, portland, oregon, usa, 2000. https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf. Accessed: 2024-06-04.

[20] Nancy Lynch Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Transactions on Computer Systems*, 2002.

List of Figures

Glossary

API **Application Programming Interface (API)** Enables a way for different components of a program to communicate with eachother

CPU **Central Processing Unit (CPU)** The main processor in a computer, which handles most tasks

Crockford Base32 **Crockford Base32** A special version of a common encoding scheme. Crockford's version is easy for humans and machines to read and type, because it is less ambiguous.

GPU **Graphics Processing Unit (GPU)** A dedicated processor intended to accelerate image processing or parallel tasks

JSON **JavaScript Object Notation (JSON)** A well-known, human-readable and standardised format to store and transmit data in

REST **Representational State Transfer (REST)** An easily scalable and well-defined software architectural style with a clear client / server relationship

SPA **Single Page Application (SPA)** A webpage, that serves dynamic content inside the current page, instead of loading completely new pages

SQL **Structured Query Language (SQL)** A language used interact with various database systems, like PostgreSQL

A. Appendices

Project management A.1

Auditor REST API A.2

Python scripts A.3

A.1. Project management

A.1.1. Definition

At start of the project, we decided upon which project management model we wanted use, how it is implemented, tracked and how it will be documented. As the goals and scope were set, we were ready to define the details. The stakeholders, project members and project roles were decided upon project launch and are defined in the titel page. The artifacts to be delivered were the following, which is the default for every thesis at the BFH:

- Source code
- Thesis report
- Poster
- Book entry
- Video
- Presentation

The following deadlines were presented by the bfh and we had to adhere:

- Till 19.04.2024: Meeting with the expert
- 28.05.2024: Delivery poster
- 10.06.2024: Delivery book entry
- 13.06.2024: Delivery video
- 13.06.2024: Delivery thesis report
- 13.06.2024: Delivery source code

- 14.06.2024: Holding presentation & Techday
- 26.06.2024: Bachelor thesis defence

Risks & mitigations

We identified the following risks for our project:

- Deadline risk
- Lack of time
- Lack of knowledge
- Absences

We had negligible risk of losing our work on artifacts, as we used Git as a software, to distribute our work done to various systems at all times. Thus our main risk become the risk of not meeting deadlines to the time or resource issues.

A.1.2. Methodology

Our approach was to keep the project management part as simple, straight forward and small as possible, as we have a very small project team and only a short project time frame.

To absorb possible shortcomings of our approach, we prioritized speed, agility and in person exchange above anything else. So decided upon using Scrum, not only as we have a lot of experience in it and we also have a certified Scrum Master in our team, but moreso because we wanted it's agility and speed. We wanted the ability to react quickly to meet approaching deadline expectations, potential failures and self set dangers. Thus we set our Sprint duration to one week and planned our sprint planning, review and retrospective meetings on each Wednesday in the whole project duration. These meetings were to be held in person with the whole project team attending, meaning professors and students, in Christian Grothoff's office at BFH's Rolex building in 2502 Biel, Höheweg 82.

A.1.3. Organization

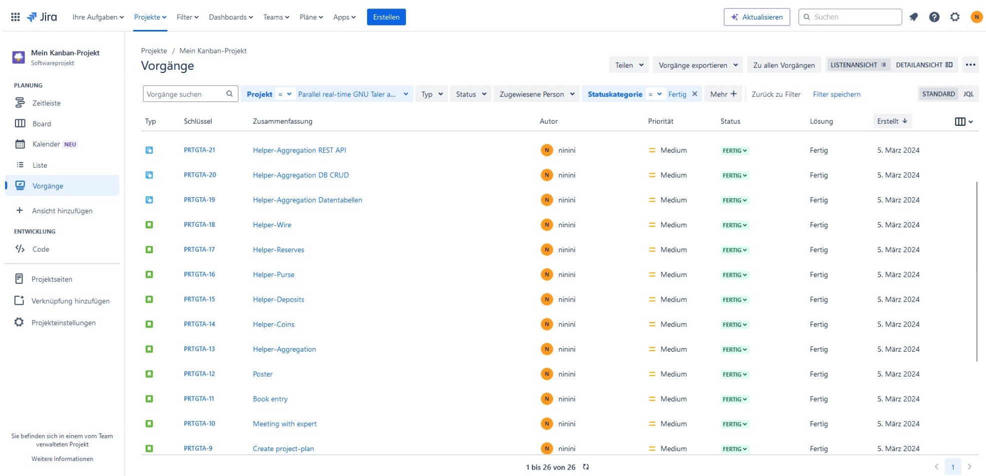

Figure A.1.: Jira backlog

After deciding the project methodology and putting the procedure in place, we quickly created our backlog. We defined each tasks with its sub tasks, specified the definition of done and the expected results. We did not go as far as to poker for story points estimates, as they did not really matter to us, because we just defined deadlines of the task's completion and worked with them, removing the need for extra project management overhead.

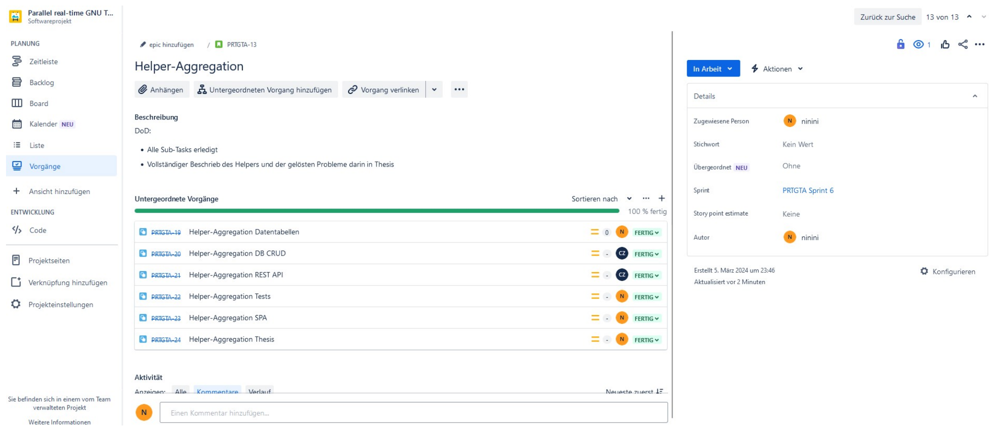

Figure A.2.: Task Helper-Aggregation

As for project communication, we agreed on using instant messengers and e-mail as part of our strategy.

A.1.4. Execution

We took note of our project meetings to keep a hold of todos. Here are some excerpts:

8.5.2024

- Remove ppdc in code
- Mark todos

15.5.2024

Poster:

- More pictures (auditor flow)
- Focus not on Taler, but the auditor and auditor architecture
- Less text
- Last section benefits, badly worded
- Don't describe the history, but the new state
- Christian checks helper wire
- Book deadline: ask BFH office
- API spec in appendices ok
- Libeufin tables: fix
- Deposit tables: fix

22.5.2024

- Taler system architecture (illustration by taler team)
- Real-time auditing for gnu taler
- Example screen poster https://uebermedien.de/wp-content/uploads/2021/02/2021-02-08-wirecard.jpg
- Example Bernie Madoff

5.6.2024

- No content: 204 / 200 or empty array
- Fixme's: in code & doc
- Remove suppress for no incidents tables
- Remove internals from api
- Endpoint sentences adjusting (remove api and sentence to endpoint)
- Code indent 2 spaces
- Wire out inconsistency
- Updates for same incident -> suppressed false
- RESTful API doc structure
- Spa frontend: dashboard1 progress & balances & details tables, dasboard2 criticals, dashboard3 lags, dashboard4 detailed state (reserve balances/active purse/-coin balances)
- Graphic adjustments: architecture
- 3.2 structure after business
- 4.2.2 remove screenshots and info structure, only text
- Remove 6.1
- Figure 2.2 label: simplified
- Auditor > auditor

A.1.5. Completion

We were able to complete the project and meet all deadline requirements. Our strategy and procedures hold strong and most importantly, managed to be successful on all set goals.

A.2. Auditor REST API

1.6. The Auditor RESTful JSON API

The API specified here follows the general conventions for all details not specified in the individual requests. The glossary defines all specific terms used in this section.

Table of Contents

Contents

1.6.1. Authentication

Each auditor instance has separate authentication settings for the private API resources of that instance.

Currently, the API supports two main authentication methods:

- `external`: With this method, no checks are done by the auditor backend. Instead, a reverse proxy / API gateway must do all authentication/authorization checks.
- `token`: With this method, the client must provide a `Authorization: Bearer $TOKEN` header, where `$TOKEN` is a secret authentication token configured for the instance which must begin with the RFC 8959 prefix.

1.6.2. Obtaining Auditor Version

This endpoint is used by merchants to obtain a list of all exchanges audited by this auditor. This may be required for the merchant to perform the required know-your-customer (KYC) registration before issuing contracts.

`GET /config`

Get the protocol version and some meta data about the auditor. This specification corresponds to `current` protocol being version **1**.

Response:

200 OK:

The auditor responds with an AuditorVersion object. This request should virtually always be successful.

Details:

```
interface AuditorVersion {
  // libtool-style representation of the Taler protocol version, see
  // https://www.gnu.org/software/libtool/manual/html_node/Versioning.html#Versioning
  // The format is "current:revision:age".  Note that the auditor
  // protocol is versioned independently of the exchange's protocol.
  version: string;

  // URN of the implementation (needed to interpret 'revision' in version).
  // @since v0, may become mandatory in the future.
  implementation?: string;

  // Return which currency this auditor is auditing for.
  currency: string;

  // EdDSA master public key of the auditor.
  auditor_public_key: EddsaPublicKey;

  // EdDSA master public key of the exchange.
  // Added in protocol v1.
  exchange_master_public_key: EddsaPublicKey;
}
```

Note

This endpoint is still experimental (and is not yet implemented at the time of this writing).

1.6.3. Deposit Confirmations

Merchants should probabilistically submit some of the deposit confirmations they receive from the exchange to auditors to ensure that the exchange does not lie about recording deposit confirmations with the exchange. Participating in this scheme ensures that in case an exchange runs into financial trouble to pay its obligations, the merchants that did participate in detecting the bad behavior can be paid out first.

`PUT /deposit-confirmation`

Submits a DepositConfirmation to the exchange. Should succeed unless the signature provided is invalid or the exchange is not audited by this auditor.

Response:

200 Ok:

The auditor responds with a DepositAudited object. This request should virtually always be successful.

403 Forbidden:

The signature on the deposit confirmation is invalid.

410 Gone:

The public key used to sign the deposit confirmation was revoked.

Details:

```
interface DepositAudited {
  // TODO: maybe change to 204 No content instead?
}
```

```
interface DepositConfirmation {

  // Hash over the contract for which this deposit is made.
  h_contract_terms: HashCode;

  // Hash over the extensions.
  h_extensions: HashCode;

  // Hash over the wiring information of the merchant.
  h_wire: HashCode;

  // Time when the deposit confirmation confirmation was generated.
  timestamp: Timestamp;

  // How much time does the merchant have to issue a refund
  // request?  Zero if refunds are not allowed.
  refund_deadline: Timestamp;

  // By what time does the exchange have to wire the funds?
  wire_deadline: Timestamp;

  // Amount to be deposited, excluding fee.  Calculated from the
  // amount with fee and the fee from the deposit request.
  amount_without_fee: Amount;

  // Array of public keys of the deposited coins.
  coin_pubs: EddsaPublicKey[];

  // Array of deposit signatures of the deposited coins.
  // Must have the same length as coin_pubs.
  coin_sigs: EddsaSignature[];

  // The Merchant's public key.  Allows the merchant to later refund
  // the transaction or to inquire about the wire transfer identifier.
  merchant_pub: EddsaPublicKey;

  // Signature from the exchange of type
  // TALER_SIGNATURE_EXCHANGE_CONFIRM_DEPOSIT.
  exchange_sig: EddsaSignature;

  // Public signing key from the exchange matching exchange_sig.
  exchange_pub: EddsaPublicKey;

  // Master public key of the exchange corresponding to master_sig.
  // Identifies the exchange this is about.
  // @deprecated since v1 (now ignored, global per auditor)
  master_pub: EddsaPublicKey;

  // When does the validity of the exchange_pub end?
  ep_start: Timestamp;

  // When will the exchange stop using the signing key?
  ep_expire: Timestamp;

  // When does the validity of the exchange_pub end?
  ep_end: Timestamp;

  // Exchange master signature over exchange_sig.
  master_sig: EddsaSignature;
}
```

Note

This endpoint is still experimental (and is not yet implemented at the time of this writing). A key open question is whether the auditor should sign the response information.

1.6.4. Monitoring API

The following entries specify how to access the results of an audit.

For most endpoints, rows may be marked as 'suppressed' to not send them again upon subsequent GET requests. To do this, a GenericAuditorMonitorPatchRequest object is used in the respective PATCH request.

Details:

```
interface GenericAuditorMonitorPatchRequest {

  // If true, subsequent GET requests will not return this element by default
  suppressed : boolean;

}
```


1.6.4.1. Fee Time Inconsistencies

This section highlights cases where validity periods associated with wire fees the exchange may charge merchants are invalid. This usually means that the validity periods given for the same type of fee are overlapping and it is thus unclear which fee really applies. This is a sign of a serious misconfiguration or data corruption as usually the exchange logic should prevent such a fee configuration from being accepted.

`GET /monitoring/fee-time-inconsistency`

Get a list of fee time inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of FeeTimeInconsistency objects. If no elements could be found, an empty array is returned

Details:

```
interface FeeTimeInconsistency {

  // Row ID of the fee in the exchange database.
  row_id : Integer;

  // Specifies the wire method for which the fee is inconsistent.
  type : string;

  // Gives the start date of the inconsistent fee.
  time : Timestamp;

  // Human readable description of the problem.
  diagnostic : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/fee-time-inconsistency/$SERIAL_ID`

This endpoint is used to suppress selected elements of fee time inconsistencies. Updates the 'suppressed' field of a fee time inconsistency element with row ID `$SERIAL_ID`.

Request:

The body must be a GenericAuditorMonitorPatchRequest.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.2. Emergencies

This endpoint is used to obtain a list of emergencies.

Emergencies are errors where the total value of coins deposited (of a particular denomination) exceeds the total value that the exchange remembers issuing. This usually means that the private keys of the exchange were compromised (stolen or factored) and subsequently used to sign coins off the books. If this happens, all coins of the respective denomination that the exchange has redeemed so far may have been created by the attacker, and the exchange would have to refund all of the outstanding coins from ordinary users. Thus, the risk exposure is the amount of coins in circulation for a particular denomination and the maximum loss for the exchange from this type of compromise.

The difference between emergencies and emergencies by count is how the auditor detected the problem: by comparing amounts, or by counting coins. Theroretically, counting coins should always detect an issue first, but given the importance of emergencies, the auditor checks both total amounts and total numbers of coins (they may differ as coins may be partially deposited).

`GET /monitoring/emergency`

Get a list of emergencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of Emergency objects. If no elements could be found, an empty array is returned

Details:

```
interface Emergency {

  // Unique row identifier
  row_id : Integer;

  // Hash of denomination public key
  denompub_h : HashCode;

  // What is the total value of all coins of this denomination that
  // were put into circulation (and thus the maximum loss the
  // exchange may experience due to this emergency).
  denom_risk : Amount;

  // What is the loss we have experienced so far (that
  // is, the amount deposited in excess of the amount
  // we issued).
  denom_loss : Amount;

  // When did the exchange start issuing coins in this the denomination.
  deposit_start : Timestamp;

  // When does the deposit period end for coins of this denomination.
  deposit_end : Timestamp;

  // What is the value of an individual coin of this denomination.
  value : Amount;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/emergency/$SERIAL_ID`

This endpoint is used to suppress select elements of emergencies. Update the 'suppressed' field of an emergency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.3. Emergencies By Count

This endpoint is used to obtain a list of emergencies by count.

Emergencies are errors where more coins were deposited than the exchange remembers issuing. This usually means that the private keys of the exchange were compromised (stolen or factored) and subsequently used to sign coins off the books. If this happens, all coins of the respective denomination that the exchange has redeemed so far may have been created by the attacker, and the exchange would have to refund all of the outstanding coins from ordinary users. Thus, the risk exposure is the amount of coins in circulation for a particular denomination and the maximum loss for the exchange from this type of compromise.

Emergencies "by count" are cases where this type of money printing was detected simply by counting the number of coins the exchange officially put into circulation and comparing it to the number of coins that were redeemed. If the number of redeemed coins is higher than the number of issued coins, the auditor reports an emergency-by-count.

`GET /monitoring/emergency-by-count`

Get a list of emergencies by count stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of EmergencyByCount objects.

Details:

```
interface EmergencyByCount {

  // Row ID of the fee in the exchange database.
  row_id : Integer;

  // Hash of the public denomination key to which the
  // emergency applies.
  denompub_h : HashCode;

  // Number of coins the exchange officially issued of this
  // denomination.
  num_issued : Integer;

  // Number of coins that were redeemed.
  num_known : Integer;

  // What is the total value of all coins of this denomination that
  // were put into circulation (and thus the maximum loss the
  // exchange may experience due to this emergency).
  risk : Amount;

  // When did the exchange start issuing coins in this the denomination.
  start : Timestamp;

  // When does the deposit period end for coins of this denomination.
  deposit_end : Timestamp;

  // What is the value of an individual coin of this denomination.
  value : Amount;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/emergency-by-count/$SERIAL_ID`

This endpoint is used to suppress select elements of emergencies by count. Update the 'suppressed' field of an emergency by count element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Request:

The body must be a GenericAuditorMonitorPatchRequest.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.4. Row Inconsistencies

This section highlights inconsistencies in a specific row of a specific table of the exchange. Row inconsistencies are reported from different sources, and largely point to some kind of data corruption (or bug). Nothing is implied about the seriousness of the inconsistency. Most inconsistencies are detected if some signature fails to validate. The affected table is noted in the 'table' field. A description of the nature of the inconsistency is noted in 'diagnostic'.

`GET /monitoring/row-inconsistency`

Get a list of row inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of RowInconsistency objects.

Details:

```
interface RowInconsistency {

  // Number of the affected row.
  row_id : Integer;

  // Name of the affected exchange table.
  row_table : string;

  // Human-readable diagnostic about what went wrong.
  diagnostic : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/row-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of row inconsistencies. Update the 'suppressed' field of a row inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.5. Reserve In Inconsistencies

This section lists cases where the exchange's and auditor's expectation of amounts transferred into a reserve differs. Basically, the exchange database states that a certain reserve was credited for a certain amount via a wire transfer, but the auditor disagrees about this basic fact. This may result in either a customer loosing funds (by being issued less digital cash than they should be) or the exchange loosing funds (by issuing a customer more digital cash than they should be).

`GET /monitoring/reserve-in-inconsistency`

Get a list of reserve in inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of ReserveInInconsistency objects.

Details:

```
interface ReserveInInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Amount the exchange expects to be in the reserve
  amount_exchange_expected : Amount;

  // Amount deposited into the reserve
  amount_wired : Amount;

  // Public key of the reserve
  reserve_pub : EddsaPublicKey;

  // Time of the deposit
  timestamp : Timestamp;

  // Account associated with the reserve
  account : string;

  // Human readable diagnostic of the problem
  diagnostic : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/reserve-in-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of reserve in inconsistencies. Update the 'suppressed' field of a reserve in inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.6. Purse Not Closed Inconsistencies

This section highlights cases, in which either payer or payee did not finish their part of a P2P payment. This caused a purse — which may contain some money — to reach its expiration date. However, the exchange failed to properly expire the purse, which means the payer did not get their money back. The cause is usually that the **taler-exchange-expire** helper is not running properly.

`GET /monitoring/purse-not-closed-inconsistencies`

Get a list of purse not closed inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of PurseNotClosedInconsistencies objects.

Details:

```
interface PurseNotClosedInconsistencies {

  // Unique row identifier.
  row_id : Integer;

  // Public key of the affected purse
  purse_pub : EddsaPublicKey;

  // Amount still in the purse, which should have been refunded
  amount : Amount;

  // When the purse expired
  expiration_date : Timestamp;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/purse-not-closed-inconsistencies/$SERIAL_ID`

This endpoint is used to suppress select elements of purse not closed inconsistencies. Update the 'suppressed' field of a purse not closed inconsistencies element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.7. Reserve Not Closed Inconsistencies

This section highlights cases, in which reserves were not closed, despite being expired. As a result, customers that wired funds to the exchange and then failed to withdraw them are not getting their money back. The cause is usually that the **taler-exchange-closer** process is not running properly.

`GET /monitoring/reserve-not-closed-inconsistency`

Get a list of reserve not closed inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of ReserveNotClosedInconsistency objects.

Details:

```
interface ReserveNotClosedInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Public key of the reserve
  reserve_pub : EddsaPublicKey;

  // Amount still in the reserve at the time of expiration
  balance : Amount;

  // Date the reserve expired
  expiration_time : Timestamp;

  // Human readable string describing the problem
  diagnostic : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

PATCH /monitoring/reserve-not-closed-inconsistency/$SERIAL_ID

This endpoint is used to suppress select elements of reserve not closed inconsistencies. Update the 'suppressed' field of a reserve not closed inconsistency element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.8. Reserve Balance Insufficient Inconsistencies

This section highlights cases where more coins were withdrawn from a reserve than the reserve contained funding for. This is a serious compromise resulting in proportional financial losses to the exchange.

GET /monitoring/reserve-balance-insufficient-inconsistency

Get a list of reserve balance insufficient inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of ReserveBalanceInsufficientInconsistency objects.

Details:

```
interface ReserveBalanceInsufficientInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Public key of the affected reserve
  reserve_pub : EddsaPublicKey;

  // Whether this inconsistency is profitable for the exchange
  inconsistency_gain : boolean;

  // Amount possibly lost or gained by the exchange
  inconsistency_amount : Amount;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/reserve-balance-insufficient-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of reserve balance insufficient inconsistencies. Update the 'suppressed' field of a reserve balance insufficient inconsistency element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.9. Invalid Signature Losses

This section lists operations that the exchange performed, but for which the signatures provided are invalid. Hence the operations are invalid and the amount involved could be a loss for the exchange (as the involved parties could successfully dispute the resulting transactions).

`GET /monitoring/bad-sig-losses`

Get a list of invalid signature losses stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.
- **operation** – A string. If specified, only returns eligible rows with this BadSigLosses.operation value. The default value is NULL which means to not filter by operaiton.
- **use_op_spec_pub** – A boolean. If true, use the value of OpSpecPub to only return eligible rows with this BadSigLosses.operation_specific_pub value. The default value is NULL.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of BadSigLosses objects.

Details:

```
interface BadSigLosses {

  // Unique row identifier
  row_id : Integer;

  // Operation performed, even though a signature was invalid
  operation : string;

  // Amount considered lost by the exchange
  loss : Amount;

  // Public key associated with an operation
  operation_specific_pub : EddsaPublicKey;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/bad-sig-losses/$SERIAL_ID`

This endpoint is used to suppress select elements of bad sig losses. Update the 'suppressed' field of a bad sig losses element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.10. Coin Inconsistencies

This section lists cases where the exchange made arithmetic errors found when looking at the transaction history of a coin. The totals sum up the differences in amounts that matter for profit/loss calculations of the exchange. When an exchange merely shifted money from customers to merchants (or vice versa) without any effects on its own balance, those entries are excluded from the total.

`GET /monitoring/coin-inconsistency`

Get a list of coin inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of CoinInconsistency objects.

Details:

```
interface CoinInconsistency {

  // Unique row identifier
  row_id : Integer;

  // The operation performed by the exchange
  operation : string;

  // Total the exchange calculated
  exchange_amount : Amount;

  // Total the auditor calculated
  auditor_amount : Amount;

  // Public key of the coin in question
  coin_pub : EddsaPublicKey;

  // Whether this arithmetic error was profitable for the exchange
  profitable : boolean;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/coin-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of coin inconsistencies. Update the 'suppressed' field of a coin inconsistency element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.11. Denominations Without Signatures

This section highlights denomination keys that lack a proper signature from the **taler-auditor-offline** tool. This may be legitimate, say in case where the auditor's involvement in the exchange business is ending and a new auditor is responsible for future denominations. So this must be read with a keen eye on the business situation.

GET /monitoring/denominations-without-sigs

Get a list of denominations without signatures stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of DenominationsWithoutSigs objects.

Details:

```
interface DenominationsWithoutSigs {

  // Unique row identifier
  row_id : Integer;

  // Hash of the denomination public key
  denompub_h : HashCode;

  // Value of each coin of the denomination that lacks
  // the auditor's signature.
  value : Amount;

  // From when the denomination key in question is valid
  start_time : Timestamp;

  // When the denomination key in question expires
  end_time : Timestamp;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

PATCH /monitoring/denominations-without-sigs/$SERIAL_ID

This endpoint is used to suppress select elements of denominations without sigs. Update the 'suppressed' field of a denominations without signatures element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.12. Misattribution In Inconsistencies

This section lists cases where the sender account record of an incoming wire transfer differs between the exchange and the bank. This may cause funds to be sent to the wrong account should the reserve be closed with a remaining balance, as that balance would be credited to the original account.

GET /monitoring/misattribution-in-inconsistency

Get a list of misattribution in inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of MisattributionInInconsistency objects.

Details:

```
interface MisattributionInInconsistency {

  // Unique row identifier in the exchange database.
  row_id : Integer;

  // Amount of money sent to the wrong account
  amount : Amount;

  // Row of the transaction in the bank database as
  // returned by the bank revenue API.
  bank_row : Integer;

  // Public key of the affected reserve
  reserve_pub : EddsaPublicKey;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/misattribution-in-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of misattribution in inconsistencies. Update the 'suppressed' field of an misattribution in inconsistency element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.13. Deposit Confirmations

This section contains a list of deposits confirmations that an exchange provided to merchants but *failed* to store in its own database. This is indicative of potential fraud by the exchange operator, as the exchange should only issue deposit confirmations after storing the respective deposit records in its database. Not storing the deposit data means that the exchange would not pay the merchant (pocketing the money) or allow the customer to double-spend the money (which is naturally also not good).

Note that entries could appear in this list also because the exchange database replication is delayed. Hence, entries that are only a few seconds old might not be indicative of an actual problem. If entries in this list are more than a few seconds old, the first thing to check is whether or not the database replication from the exchange is working properly.

`GET /monitoring/deposit-confirmations`

Get a list of deposit confirmations stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of DepositConfirmations objects.

Details:

```
interface DepositConfirmations {

  // Row id in the exchange database
  deposit_confirmation_serial_id : Integer;

  // Hash over the contract for which this deposit is made.
  h_contract_terms : HashCode;

  // Hash over the policy concerning this deposit
  h_policy : HashCode;

  // Hash over the wiring information of the merchant.
  h_wire : HashCode;

  // Time when the deposit confirmation confirmation was generated.
  exchange_timestamp : Timestamp;

  // How much time does the merchant have to issue a refund
  // request?  Zero if refunds are not allowed.
  refund_deadline : Timestamp;

  // By what time does the exchange have to wire the funds?
  wire_deadline : Timestamp;

  // Amount to be deposited, excluding fee.  Calculated from the
  // amount with fee and the fee from the deposit request.
  total_without_fee : Amount;

  // Array of public keys of the deposited coins.
  coin_pubs : EddsaPublicKey[];

  // Array of deposit signatures of the deposited coins.
  // Must have the same length as coin_pubs.
  coin_sigs : EddsaSignature[];

  // The Merchant's public key.  Allows the merchant to later refund
  // the transaction or to inquire about the wire transfer identifier.
  merchant_pub : EddsaPublicKey;

  // Signature from the exchange of type
  // TALER_SIGNATURE_EXCHANGE_CONFIRM_DEPOSIT.
  exchange_sig : EddsaSignature;

  // Public signing key from the exchange matching exchange_sig.
  exchange_pub : EddsaPublicKey;

  // Exchange master signature over exchange_sig.
  master_sig : EddsaSignature;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/deposit-confirmations/$SERIAL_ID`

This endpoint is used to suppress select elements of deposit confirmations. Update the 'suppressed' field of an deposit confirmations element with row ID `$SERIAL_ID`, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.14. Denomination Key Validity Withdraw Inconsistencies

This section highlights cases, where denomination keys were used to sign coins withdrawn from a reserve before the denomination was valid or after it was already expired for signing. This doesn't exactly imply any financial loss for anyone, it is mostly weird and may have affected the fees the customer paid.

`GET /monitoring/denomination-key-validity-withdraw-inconsistency`

Get a list of denomination key validity withdraw inconsistencies stored by the auditor. The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of DenominationKeyValidityWithdrawInconsistency objects. If no elements could be found, an empty array is returned

Details:

```
interface DenominationKeyValidityWithdrawInconsistency {

  // Unique row identifier
  row_id : Integer;

  // When the withdrawal took place
  execution_date : Timestamp;

  // Public key of the reserve affected
  reserve_pub : EddsaPublicKey;

  // Hash of the denomination public key involved in the withdrawal
  denompub_h : HashCode;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/denomination-key-validity-withdraw-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of denomination key validity withdraw inconsistencies. Update the 'suppressed' field of a denomination key validity withdraw inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.15. Amount Arithmetic Inconsistencies

This endpoint is used to obtain a list of amount arithmetic inconsistencies.

This section lists cases where the arithmetic of the exchange involving amounts disagrees with the arithmetic of the auditor. Disagreements imply that either the exchange made a loss (sending out too much money), or screwed a customer (and thus at least needs to fix the financial damage done to the customer). The profitable column is set to true if the arithmetic problem was be determined to be profitable for the exchange, false if the problem resulted in a net loss for the exchange.

`GET /monitoring/amount-arithmetic-inconsistency`

Get a list of amount arithmetic inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20. 70
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of AmountArithmeticInconsistency objects. If no elements could be found, an empty array is returned

Details:

```
interface AmountArithmeticInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Name of the arithmetic operation performed
  operation : string;

  // Amount according to the exchange
  exchange_amount : Amount;

  // Amount according to the auditor
  auditor_amount : Amount;

  // Whether the miscalculation is profitable for the exchange
  profitable : boolean;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/amount-arithmetic-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of amount arithmetic inconsistencies. Update the 'suppressed' field of an amount arithmetic inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.16. Wire Format Inconsistencies

This section highlights cases where the wire transfer subject was used more than once and is thus not unique. This indicates a problem with the bank's implementation of the revenue API, as the bank is supposed to warrant uniqueness of wire transfer subjects exposed via the revenue API (and bounce non-unique transfers).

`GET /monitoring/wire-format-inconsistency`

Get a list of wire format inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of WireFormatInconsistency objects. If no elements could be found, an empty array is returned

Details:

```
interface WireFormatInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Amount that was part of the wire
  amount : Amount;

  // Offset of the duplicate wire transfer subject
  // in the bank database according to the revenue API.
  wire_offset : Integer;

  // True if this diagnostic was suppressed.
  diagnostic : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/wire-format-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of wire format inconsistencies. Update the 'suppressed' field of a wire format inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.17. Refreshes Hanging

This section highlights cases, where a coin was melted but the reveal process was not finished by the wallet. Usually, a wallet will do both requests in rapid succession to refresh a coin. This might happen, even if the exchange is operating correctly, if a wallet goes offline after melting. However, after some time wallets should in most cases come back online and finish the operation. If many operations are hanging, this might be indicative of a bug (exchange failing on reveal, or wallets not implementing refresh correctly).

`GET /monitoring/refreshes-hanging`

Get a list of refreshes hanging stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of RefreshesHanging objects. If no elements could be found, an empty array is returned

Details:

```
interface RefreshesHanging {

  // Unique row identifier
  row_id : Integer;

  // Amount in coin not found in the exchange
  amount : Amount;

  // Public key of coin
  coin_pub : EddsaPublicKey;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/refreshes-hanging/$SERIAL_ID`

This endpoint is used to suppress select elements of refreshes hanging. Update the 'suppressed' field of a refreshes hanging element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.18. Closure Lags

This endpoint is used to obtain a list of closure lags.

A closure lag happens if a reserve should have closed a reserve and wired (remaining) funds back to the originating account, but did not do so on time. Significant lag may be indicative of fraud, while moderate lag is indicative that the systems may be too slow to handle the load. Small amounts of lag can occur in normal operation.

If closure lag is experienced, the administrator should check that the **taler-exchange-closer** component is operating correctly.

`GET /monitoring/closure-lags`

Get a list of closure lags stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of ClosureLags objects. If no elements could be found, an empty array is returned

Details:

```
interface ClosureLags {

  // Unique row identifier
  row_id : Integer;

  // Amount of money left in the reserve
  amount : Amount;

  // When should the reserve have been closed
  deadline : Timestamp;

  // The wire transfer identifier
  wtid : HashCode;

  // payto URI (RFC 8905) of the account that
  // should have been credited.
  account : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/closure-lags/$SERIAL_ID`

This endpoint is used to suppress select elements of closure lags. Update the 'suppressed' field of a closure lags element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.19. Wire Out Inconsistencies

This section highlights cases where the exchange wired a different amount to a destination account than the auditor expected.

`GET /monitoring/wire-out-inconsistency`

Get a list of wire out inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of WireOutInconsistency objects. If no elements could be found, an empty array is returned

Details:

```
interface WireOutInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Account money was wired to
  destination_account : string;

  // How much was supposed to be wired according to the auditor.
  expected : Amount;

  // The amount the exchange claims to have wired.
  claimed : Amount;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/wire-out-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of wire out inconsistencies. Update the 'suppressed' field of a wire out inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.20. Reserve Balance Summary Wrong Inconsistencies

This section highlights cases, where the exchange's and auditors' expectation of the amount of money left in a reserve differs.

`GET /monitoring/reserve-balance-summary-wrong-inconsistency`

Get a list of reserve balance summary wrong inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of ReserveBalanceSummaryWrongInconsistency objects. If no elements could be found, an empty array is returned

Details:

```
interface ReserveBalanceSummaryWrongInconsistency {

  // Unique row identifier
  row_id : Integer;

  // Public key of the reserve affected
  reserve_pub : EddsaPublicKey;

  // Amount of summary the exchange calculated
  exchange_amount : Amount;

  // Amount of summary the auditor calculated
  auditor_amount : Amount;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/reserve-balance-summary-wrong-inconsistency/$SERIAL_ID`

This endpoint is used to suppress select elements of reserve balance summary wrong inconsistencies. Update the 'suppressed' field of a reserve balance summary wrong inconsistency element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.4.21. Row Minor Inconsistencies

The section highlights inconsistencies where a row in an exchange table has a value that is does not satisfy expectations (such as a malformed signature). These are cause for concern, but not necessarily point to a monetary loss (yet).

`GET /monitoring/row-minor-inconsistencies`

Get a list of row minor inconsistencies stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **return_suppressed** – A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.

With the default settings, the endpoint returns at most the 20 latest elements that are not suppressed.

Response:

200 OK:

The auditor responds with a top level array of RowMinorInconsistencies objects. If no elements could be found, an empty array is returned

Details:

```
interface RowMinorInconsistencies {

  // Number of the row in the affected table
  row_id : Integer;

  // The row number in the affected table
  row_table : Integer;

  // Human readable string describing the problem
  diagnostic : string;

  // True if this diagnostic was suppressed.
  suppressed : boolean;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

`PATCH /monitoring/row-minor-inconsistencies/$SERIAL_ID`

This endpoint is used to suppress select elements of row minor inconsistencies. Update the 'suppressed' field of a row minor inconsistencies element with row_id $SERIAL_ID, according to GenericAuditorMonitorPatchRequest, stored by the auditor.

Response:

204 No Content:

The element has been updated.

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5. Monitoring Auditor Status

The following entries specify how to access information the auditor keeps to properly perform audits. These tables do not contain inconsistencies, instead they store information about balances, reserves, purses etc. Values in these tables should not differ from their respective exchanges' version.

1.6.5.1. Balances

Returns the various balances the auditor tracks for the exchange, such as coins in circulation, fees earned, losses experienced, etc.

`GET /monitoring/balances`

Get a list of balances stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
- **balance_key** – a string identifying a balance. If specified, only returns elements with this exact key. The default value is NULL.

With the default settings, the endpoint returns at most the 20 latest elements.

Response:

200 OK:

The auditor responds with a top level array of Balances objects. If no elements could be found, an empty array is returned

Details:

```
interface Balances {

  // Unique row identifier
  row_id : Integer;

  // String identifying a balance
  balance_key : string;

  // Amount of the balance
  balance_value : Amount;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5.2. Historic Denomination Revenue

This endpoint is used to obtain a list of historic denomination revenue, that is the profits and losses an exchange has made from coins of a particular denomination where the denomination is past its (deposit) expiration and thus all values are final.

`GET /monitoring/historic-denomination-revenue`

Get a list of historic denomination revenue stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.

With the default settings, the endpoint returns at most the 20 latest elements.

Response:

200 OK:

The auditor responds with a top level array of HistoricDenominationRevenue objects. If no elements could be found, an empty array is returned

Details:

```
interface HistoricDenominationRevenue {

  // Unique row identifier
  row_id : Integer;

  // Hash code of the denomination public key involved
  denom_pub_hash : HashCode;

  // Time when the denomination expired and thus the revenue
  // was computed.
  revenue_timestamp : Timestamp;

  // Total fee revenue the exchange earned from coins of this
  // denomination.
  revenue_balance : Amount;

  // Total losses the exchange experienced from this denomination
  // (this basically only happens if someone was able to forge
  // denomination signatures). So non-zero values are indicative
  // of a serious problem.
  loss_balance : Amount;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5.3. Denomination Pending

This endpoint is used to obtain a list of balances for denominations that are still active, that is coins may still be deposited (or possibly even withdrawn) and thus the amounts given are not final.

GET /monitoring/denomination-pending

Get a list of denomination pending stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.

With the default settings, the endpoint returns at most the 20 latest elements.

Response:

200 OK:

The auditor responds with a top level array of DenominationPending objects. If no elements could be found, an empty array is returned

Details:

```
interface DenominationPending {

  // Unique row identifier
  row_id : Integer;

  // Hash of the denomination public key
  denom_pub_hash : HashCode;

  // Total value of coins remaining in circulation (excluding
  // the value of coins that were recouped, those are always
  // just under recoup_loss).
  denom_balance : Amount;

  // Total value of coins redeemed that exceeds the amount we
  // put into circulation. Basically, this value grows if we
  // wanted to reduce denom_balance (because a coin was deposited)
  // but we could not because the denom_balance was already zero.
  denom_loss : Amount;

  // Total number of coins of this denomination that were
  // put into circulation.
  num_issued : Integer;

  // Total value of the coins put into circulation.
  denom_risk : Amount;

  // Losses the exchange had from this denomination due to coins
  // that were recouped (after the denomination was revoked).
  recoup_loss : Amount;

}
```


> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5.4. Historic Reserve Summary

This section summarizes historic profits an exchange made from reserves and associated reserve-specific fees.

GET /monitoring/historic-reserve-summary

Get a list of historic reserve summary stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.

With the default settings, the endpoint returns at most the 20 latest elements.

Response:

200 OK:

The auditor responds with a top level array of HistoricReserveSummary objects. If no elements could be found, an empty array is returned

Details:

```
interface HistoricReserveSummary {

  // Unique row identifier
  row_id : Integer;

  // From when the summary starts
  start_date : Timestamp;

  // When the summary ends
  end_date : Timestamp;

  // Profits the exchange charged for the reserve
  reserve_profits : Amount;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5.5. Reserves

This endpoint is used to obtain a list of open reserves that the auditor is currently tracking balances for.

GET /monitoring/reserves

Get a list of reserves stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:

- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.

With the default settings, the endpoint returns at most the 20 latest elements.

Response:

200 OK:

The auditor responds with a top level array of Reserves objects. If no elements could be found, an empty array is returned

Details:

```
interface Reserves {

  // Unique row identifier
  auditor_reserves_rowid : Integer;

  // Public key of the reserve
  reserve_pub : EddsaPublicKey;

  // Amount in the balance
  reserve_balance : Amount;

  // Reserve losses are incurred if (a) a reserve is
  // incorrectly credited from a recoup for a non-revoked
  // coin, or (b) if the exchange allowed more digital cash
  // to be withdrawn from a reserve than the balance of the
  // reserve should have permitted.  FIXME: We may want to
  // distinguish these two cases in the future.
  reserve_loss : Amount;

  // Amount earned by charging withdraw fees
  withdraw_fee_balance : Amount;

  // Amount earned by charging a closing fee on the reserve
  close_fee_balance : Amount;

  // Total purse fees earned from this reserve
  purse_fee_balance : Amount;

  // Total reserve open fees earned from the reserve
  open_fee_balance : Amount;

  // Total reserve history fees earned from this reserve
  history_fee_balance : Amount;

  // When the purse expires
  expiration_date : Timestamp;

  // Who created the account
  origin_account : string;

}
```

> **Note**
>
> This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5.6. Purses

This endpoint is used to obtain information about open purses.

GET /monitoring/purses

Get a list of purses stored by the auditor.

The following query parameters are optional, and can be used to customise the response:

Request:

Query Parameters:
- **limit** – A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
- **offset** – An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.

With the default settings, the endpoint returns at most the 20 latest elements.

Response:

200 OK:

The auditor responds with a top level array of Purses objects. If no elements could be found, an empty array is returned

Details:

```
interface Purses {

  // Unique row identifier
  auditor_purses_rowid : Integer;

  // Public key of the purse
  purse_pub : EddsaPublicKey;

  // Amount currently stored in the purse
  balance : Amount;

  // Amount the purse is intended for / the maximum amount that can be in the purse
  target : Amount;

  // When the purse expires
  expiration_date : Timestamp;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.5.7. Progress

This section contains information about the auditing progress an auditor has made.

`GET /monitoring/progress`

Get the progress stored by the auditor.

Response:

200 OK:

The auditor responds with a top level array of Progress objects. If no elements could be found, an empty array is returned

Details:

```
interface Progress {

  // Key associated with a given progress point
  progress_key : String;

  // How much of the exchanges data has been processed so far
  progress_offset : Integer;

}
```

Note

This endpoint is still experimental. The endpoint will be further developed as needed.

1.6.6. Complaints

This endpoint is used by the wallet or merchants to submit proof of misbehavior of an exchange to the auditor.

Note

To be designed and implemented.

`PUT /complain`

Complain about misbehavior to the auditor.

Previous
1.5. Wallet-Core API Documentation

Next
1.7. Backup and Synchronization RESTful API

A.3. Python Scripts

```
import os
import re

dcm = """
.. _deposit-confirmation:

=====================
Deposit Confirmations
=====================

Merchants should probabilistically submit some of the deposit
confirmations they receive from the exchange to auditors to ensure
that the exchange does not lie about recording deposit confirmations
with the exchange. Participating in this scheme ensures that in case
an exchange runs into financial trouble to pay its obligations, the
merchants that did participate in detecting the bad behavior can be
paid out first.

.. http:put:: /deposit-confirmation

  Submits a `DepositConfirmation` to the exchange. Should succeed
  unless the signature provided is invalid or the exchange is not
  audited by this auditor.

  **Response:**

  :http:statuscode:`200 Ok`:
        The auditor responds with a `DepositAudited` object.
        This request should virtually always be successful.
  :http:statuscode:`403 Forbidden`:
        The signature on the deposit confirmation is invalid.
  :http:statuscode:`410 Gone`:
        The public key used to sign the deposit confirmation
        was revoked.

  **Details:**

  .. ts:def:: DepositAudited

    interface DepositAudited {
      // TODO: maybe change to ``204 No content`` instead?
    }

  .. ts:def:: DepositConfirmation

    interface DepositConfirmation {

      // Hash over the contract for which this deposit is made.
      h_contract_terms: HashCode;

      // Hash over the extensions.
      h_extensions: HashCode;

      // Hash over the wiring information of the merchant.
      h_wire: HashCode;

      // Time when the deposit confirmation confirmation was generated.
      timestamp: Timestamp;

      // How much time does the merchant have to issue a refund
```

```
      // request?  Zero if refunds are not allowed.
      refund_deadline: Timestamp;

      // By what time does the exchange have to wire the funds?
      wire_deadline: Timestamp;

      // Amount to be deposited, excluding fee.  Calculated from the
      // amount with fee and the fee from the deposit request.
      amount_without_fee: Amount;

      // Array of public keys of the deposited coins.
      coin_pubs: EddsaPublicKey[];

      // Array of deposit signatures of the deposited coins.
      // Must have the same length as ``coin_pubs``.
      coin_sigs: EddsaSignature[];

      // The Merchant's public key.  Allows the merchant to later refund
      // the transaction or to inquire about the wire transfer identifier.
      merchant_pub: EddsaPublicKey;

      // Signature from the exchange of type
      // ``TALER_SIGNATURE_EXCHANGE_CONFIRM_DEPOSIT``.
      exchange_sig: EddsaSignature;

      // Public signing key from the exchange matching ``exchange_sig``.
      exchange_pub: EddsaPublicKey;

      // Master public key of the exchange corresponding to ``master_sig``.
      // Identifies the exchange this is about.
      // @deprecated since v1 (now ignored, global per auditor)
      master_pub: EddsaPublicKey;

      // When does the validity of the exchange_pub end?
      ep_start: Timestamp;

      // When will the exchange stop using the signing key?
      ep_expire: Timestamp;

      // When does the validity of the exchange_pub end?
      ep_end: Timestamp;

      // Exchange master signature over ``exchange_sig``.
      master_sig: EddsaSignature;
    }

  .. note::

    This API is still experimental (and is not yet implemented at the
    time of this writing). A key open question is whether the auditor
    should sign the response information.

"""

dcm_del = """

This API is used by the auditor to delete an audited deposit confirmation.

.. http:delete:: /deposit-confirmation/$SERIAL_ID

  Delete deposit confirmation entry with given serial_id.

```

```
  **Response:**

  :http:statuscode:`204 No content`:
    The deposit confirmation was deleted.

  :http:statuscode:`401 Unauthorized`:
    Unauthorized request.

  :http:statuscode:`404 Not found`:
    The deposit confirmation was unknown.

  .. note::

    This API is still experimental (and is not yet implemented at the
    time of this writing).
"""

spa_api = f"""
.. _spa-api:

---------------------------
Single Page Application API
---------------------------

The following entries specify how to access the results of an audit.

For most endpoints, rows may be marked as 'suppressed', to not send them again upon
     subsequent GET requests.
To do this, a :ts:type:`GenericUpdate` object may be used.

**Details:**

  .. ts:def:: GenericUpdate

    interface GenericUpdate {{

        // the row_id of a respective table that should be changed
        row_id : Integer;

        suppressed : boolean;

        // unused
        ancient? : boolean;

    }}

"""

en = {

    "u_int64" : "Integer",
    "taler_amount" : "Amount",
    "boolean" : "boolean",
    "text" : "string"

}


descriptions = {
   "fee-time-inconsistency" : """

""",
```

```
  "amount-arithmetic-inconsistency" : """

""",
  "closure-logs" : """

""",
"bad-sig-losses" : """
This table tracks the amount of money lost because of bad signatures.
""",
}






def repl(tp):


   if tp not in en:
    return "TODO"
   else:
    return en[tp]

def guessBYTEA(prop):
   # prop is the properties name, like "denompub_h"

  if prop == "row_id":
     return "Integer"

  if prop.endswith("_h") or prop.endswith("_hash") or prop.startswith("h_"):
    return "HashCode"

  if "time" in prop or "ends" in prop or "start" in prop or "_date" in prop or "_end" in prop or "deadline" in prop or "expire" in prop or prop.endswith("_from"):
    return "Timestamp"

  if "_pub" in prop:
    return "EddsaPublicKey"

  if "_sig" in prop:
    return "EddsaSignature"

  if "diagnostic" in prop or "operation" in prop:
      return "string"

  if prop == "destination_account" or prop == "account" or prop == "type":
     return "string"

  if "num_" in prop or "offset" in prop or "row" in prop or prop.endswith("_id"):
     return "Integer"

  if "wtid" == prop:
     return "Integer"

  return "TODO"

def doc_upd(a):
    w = a[0]

    sc = a[2]
    ssc = a[3]
```

```
    kc = a[4]
    cc = a[5]
    s = a[6]

    s_plur = a[7]

    template = f"""

This API is used to suppress select elements of {s_plur}

.. http:patch:: /{kc}

  Update the 'suppressed' field of an {s} element according to :ts:type:`GenericUpdate`,
      stored by the auditor.

  **Response:**

  :http:statuscode:`202 Accepted`:
    The element has been accepted for processing.

  .. note::

    This API is still experimental. The API will be further developed as needed.


"""

    return template



def doc_get(a):

    w = a[0]

    sc = a[2]
    ssc = a[3]
    kc = a[4]
    cc = a[5]
    s = a[6]
    s_plur = a[7]
    s_plur_caps = a[8]

    addendum = ""

    if kc == "bad-sig-losses":
      addendum = """:query operation: A string. If specified, only returns eligible rows
           with this :ts:type:`BadSigLosses`.operation value. The default value is NULL.
  :query use_op_spec_pub: A boolean. If true, use the value of :ts:type:`OpSpecPub` to
      only return eligible rows with this :ts:type:`BadSigLosses`.operation_specific_pub
      value. The default value is NULL.
"""
    if kc == "balances":
       addendum = """:query balance_key: a string identifying a balance. If specified,
           only returns elements with this exact key. The default value is NULL.
"""

    s_len = len(f"{s_plur_caps}")
    cov = "-" * s_len

    tbl_con = ""

```

```
    for x in w:
        tbl_con += "\n\t" + x + " : " + w[x]  + ";\n"

    template = f"""

.. {kc}-list:

{s_plur_caps}
{cov}

This API is used to obtain a list of {s_plur}

.. http:get:: /{kc}

  Get a list of {s_plur} stored by the auditor.

  The following query parameters are optional, and can be used to customise the response:

  **Request:**

  :query limit: A signed integer, indicating how many elements relative to the offset query parameter should be returned. The default value is -20.
  :query offset: An unsigned integer, indicating from which row onward to return elements. The default value is INT_MAX.
  :query return_suppressed: A boolean. If true, returns all eligible rows, otherwise only returns eligible rows that are not suppressed. The default value is false.
  {addendum}

  The default values, thus, return at max the 20 latest elements that are not suppressed.

  **Response:**

  :http:statuscode:`200 OK`:
    The auditor responds with a top level array of :ts:type:`{cc}` objecs.

  :http:statuscode:`403 Forbidden`:
    No or bad Bearer token provided.

  :http:statuscode:`404 Not Found`:
    No elements could be found.

  **Details:**

  .. ts:def:: {cc}

    interface {cc} {{

        { tbl_con }

    }}

  .. note::

    This API is still experimental. The API will be further developed as needed.


"""

    return template


def main():
```

```

f = open("doc.txt","w+")

f.write(dcm)

f.write(spa_api)

amalgamation = list()

directory = os.fsencode("sql")

for file in os.listdir(directory):



  words = {}

  name = os.fsdecode(file)
  path = os.fsdecode(directory)

  if name.find("DS_Store") != -1:
      continue

  nm = name.removesuffix(".sql")
  comp = list( filter(lambda x: x != "0002-auditor",nm.split('_')) )


  sql = open(path + '/' +  name, 'r', encoding='utf-8', errors='ignore')

  lines = sql.readlines()

  i = 0
  for line in lines:
      #find point of interest
      if (line.find("CREATE␣TABLE") < 0):
          i += 1
          continue
      else:
          i += 1
          # skips one, but that is ok
          exit = 0
          for x in range(i,len(lines) - 1):
              sql = lines[x]

              if (sql.find(");") >= 0):
                  exit = 1

              if (exit == 0):
                  sql = re.sub(r'[^\w\s]', '', sql)

                  if (sql != '\n'):

                      dingdong = sql.split('␣')

                      bloop = list(filter(lambda x: x != '',dingdong))

                      #print(bloop)

                      subst = repl(bloop[1].strip().lower())

                      if subst == "TODO":

```

```
                            subst = guessBYTEA(bloop[0].strip().lower())

                            words[bloop[0].strip().lower()] = subst
                          else:
                            words[bloop[0].strip().lower()] = subst


      sc = "_".join(comp)
      ssc = "_".join(map(str.upper,comp))
      kc = "-".join(comp)
      cc = "".join(map(str.capitalize,comp))
      s = " ".join(comp)

      for i, n in enumerate(comp):
        if comp[i] == "inconsistency":
            comp[i] = "inconsistencies"
        if comp[i] == "emergency":
            comp[i] = "emergencies"

      s_plur = " ".join(comp)
      s_plur_caps = " ".join(map(str.capitalize,comp))

      tpl = (words, comp, sc, ssc, kc, cc, s, s_plur, s_plur_caps)

      amalgamation.append(tpl)
      f.write(doc_get(tpl))

      f.write(doc_upd(tpl))

      if (kc == "deposit-confirmations"):
         f.write(dcm_del)

    f.close()












if __name__ == "__main__":
    main()
```

```
aggregation = [ "coin_history",
    "coin_deposits",
    "refresh_commitments",
    "purse_deposits",
    "purse_decision",
    "refunds",
    "recoup_refresh",
    "recoup",
    "reserves_open_deposits",
    "known_coins",

    "batch_deposits",
    "wire_targets",
    "partners",
    "purse_requests",
```

```
    "purse_decision",
    "purse_deposits",
    "coin_deposits",
    "refresh_revealed_coins",
    "reserves_out",
    "reserves",
    "refresh_commitments",
    "recoup",
    "recoup_refresh",
    "coin_history",
    "refunds",
    "reserves_open_deposits",
    "known_coins",

    "aggregation_tracking",
    "batch_deposits",
"coin_deposits",
"wire_targets",
"known_coins",

"wire_out",
"wire_out",
"wire_targets"
    ]

coins = [
"denomination_revocations",

"known_coins",

"refresh_commitments",
"refresh_revealed_coins",

"purse_deposits",
"known_coins",

"auditor_denom_sigs",
"auditors",


"reserves_out",
"reserves",

"refunds",
"batch_deposits",
"coin_deposits",
"known_coins",

"purse_decision",
"purse_requests",
"purse_merges",
"recoup_refresh",
"refresh_revealed_coins",
"refresh_commitments",
"known_coins",

"recoup",
"known_coins",
"reserves_out",
"reserves",

"refresh_commitments",
```

```
"known_coins",

"coin_deposits",
"batch_deposits",
"wire_targets",
"known_coins",

"purse_deposits",
"partners",
"purse_merges",
"purse_requests",
"known_coins",

]

deposits = [
"coin_deposits",
"batch_deposits",
"known_coins",

"wire_targets"

]


# auditor_purses has been deliberately removed
purses = [
"global_fee",
"purse_requests",
"purse_deposits",
"partners",
"purse_merges",
"purse_requests",
"known_coins",

"account_merges",
"purse_requests",
"purse_merges",
"purse_decision",
"purse_merges",
"purse_requests",
"partners"
]

reserves = [
"denomination_revocations",

"wire_fee",
"reserves_in",
"reserves",
"wire_targets",
"reserves_out",
"reserves",

"recoup",
"known_coins",
"reserves_out",
"reserves",

"reserves_open_requests",
"reserves_close",
"wire_targets",
```

```
"reserves",
"purse_decision",
"purse_requests",
"purse_merges",
"purse_requests",
"purse_merges"
]

wire = [
"aggregation_tracking",
"profit_drains",
"wire_out",
"wire_targets",
"reserves_in",
"reserves",
"wire_targets",
"reserves_close",
"wire_targets",
"reserves"


]


def main():
    c = 0
    for l in [(aggregation, "auditor_wake_aggregation_helper_trigger"), (coins,
         "auditor_wake_coins_helper_trigger"), (purses,
         "auditor_wake_purses_helper_trigger"), (deposits,
         "auditor_wake_deposits_helper_trigger"), (reserves,
         "auditor_wake_reserves_helper_trigger"),
         (wire,"auditor_wake_wire_helper_trigger")]:

        compr = list(set(l[0]))
        i = 0

        for tbl in compr:
            str = f"""
CREATE OR REPLACE TRIGGER auditor_exchange_notify_helper_{l[1].split("_")[2]}{i}
    AFTER INSERT ON exchange.{tbl}
    EXECUTE FUNCTION {l[1]}();
        """

            print(str)

            i = i + 1

        c = c + 1




if __name__ == "__main__":
    main()
```

```
import time
import os
import re

license = """
/*
   This file is part of TALER
```

```
   Copyright (C) 2024 Taler Systems SA

   TALER is free software; you can redistribute it and/or modify it under the
   terms of the GNU General Public License as published by the Free Software
   Foundation; either version 3, or (at your option) any later version.

   TALER is distributed in the hope that it will be useful, but WITHOUT ANY
   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR
   A PARTICULAR PURPOSE.  See the GNU General Public License for more details.

   You should have received a copy of the GNU General Public License along with
   TALER; see the file COPYING.  If not, see <http://www.gnu.org/licenses/>
 */
"""


pack_json =  {
        "bigint" : "GNUNET_JSON_pack_int64",
        "integer" : "GNUNET_JSON_pack_int64",
        "int8" : "GNUNET_JSON_pack_int32",
        "bytea" : "GNUNET_JSON_pack_data_auto",
        "taler_amount" : "TALER_JSON_pack_amount",
        "boolean" : "GNUNET_JSON_pack_bool",
        "varchar" : "GNUNET_JSON_pack_string",
        "text" : "GNUNET_JSON_pack_string"
        }

def pkjs(param, t):

    pr = pack_json[t]

    match t:
        case "taler_amount":
            return pr + f"(\"{param}\",␣&dc->{param})"

    return pr + f"(\"{param}\",␣dc->{param})"

#this amount needs three arguments
spec_json = {

        "bigint" : "GNUNET_JSON_spec_int64",
        "integer" : "GNUNET_JSON_spec_int64",
        "int8" : "GNUNET_JSON_spec_int32",
        "bytea" : "GNUNET_JSON_spec_fixed_auto",
        "taler_amount" : "TALER_JSON_spec_amount",
        "boolean" : "GNUNET_JSON_spec_bool",
        "varchar" : "GNUNET_JSON_spec_string",
        "text" : "GNUNET_JSON_spec_string"

}

def spjs(param, t):

    pr = spec_json[t]

    match t:
        case "taler_amount":
            return pr + f"(\"{param}\",␣TAH_currency,␣&dc.{param})"
        case "varchar":
            return pr + f"(\"{param}\",␣(const␣char␣**)␣&dc.{param})"

    return pr + f"(\"{param}\",␣&dc.{param})"
```

```

spec_pq = {

        "bigint" : "GNUNET_PQ_result_spec_int64",
        "integer" : "GNUNET_PQ_result_spec_int64",
        "int8" : "GNUNET_PQ_result_spec_int32",
        "bytea" : "GNUNET_PQ_result_spec_auto_from_type",
        "taler_amount" : "TALER_PQ_RESULT_SPEC_AMOUNT",
        "boolean" : "GNUNET_PQ_result_spec_bool",
        "varchar" : "GNUNET_PQ_result_spec_string",
        "text" : "GNUNET_PQ_result_spec_string"

}

def sppq(param, t):

    pr = spec_pq[t]

    return pr + f"(\"{param}\",␣␣&dc.{param})"

query_pq = {

        "bigint" : "GNUNET_PQ_query_param_int64",
        "integer" : "GNUNET_PQ_query_param_int64",
        "int8" : "GNUNET_PQ_query_param_int32",
        "bytea" : "GNUNET_PQ_query_param_auto_from_type",
        "taler_amount" : "TALER_PQ_query_param_amount",
        "boolean" : "GNUNET_PQ_query_param_bool",
        "varchar" : "GNUNET_PQ_query_param_string",
        "text" : "GNUNET_PQ_query_param_string"

}

def qupq(param, t):

    pr = query_pq[t]

    match t:
        case "string":
            return pr + f"(dc->{param})"
        case "boolean":
            return pr + f"(dc->{param})"
        case "taler_amount":
            return pr + f"(pg->conn,␣&dc->{param})"
        case "bytea":
            return pr + f"(&dc->{param})"

    return pr + f"(&dc->{param})"


c_types = {

        "bigint" : "int64_t",
        "integer" : "int64_t",
        "int8" : "int32_t",
        "bytea" : "TYPE",
        "taler_amount" : "struct␣TALER_Amount",
        "boolean" : "bool",
        "varchar" : "char␣*",
        "text" : "char␣*"

}
```

```
def taler_auditor_httpd_xyz_put_new(snake_case, screaming_snake_case, kebab_case,
    camelCase, pl):

    ret = f"""
    {license}


#include "platform.h"
#include <gnunet/gnunet_util_lib.h>
#include <gnunet/gnunet_json_lib.h>
#include <jansson.h>
#include <microhttpd.h>
#include <pthread.h>
#include "taler_json_lib.h"
#include "taler_mhd_lib.h"
#include "taler-auditor-httpd.h"
#include "taler-auditor-httpd_{kebab_case}-put.h"

/**
* We have parsed the JSON information about the {kebab_case}, do some
* basic sanity checks and then execute the
* transaction.
*
* @param connection the MHD connection to handle
* @param dc information about the {kebab_case}
* @return MHD result code
*/
static MHD_RESULT
process_inconsistency (
  struct MHD_Connection *connection,
  const struct TALER_AUDITORDB_{camelCase} *dc)
{{

  enum GNUNET_DB_QueryStatus qs;

  if (GNUNET_SYSERR ==
      TAH_plugin->preflight (TAH_plugin->cls))
  {{
    GNUNET_break (0);
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_INTERNAL_SERVER_ERROR,
                                       TALER_EC_GENERIC_DB_SETUP_FAILED,
                                       NULL);
  }}

  /* execute transaction */
  qs = TAH_plugin->insert_{snake_case} (TAH_plugin->cls,
                                        dc);
  if (0 > qs)
  {{
    GNUNET_break (GNUNET_DB_STATUS_HARD_ERROR == qs);
    TALER_LOG_WARNING (
      "Failed to store /{kebab_case} in database\n");
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_INTERNAL_SERVER_ERROR,
                                       TALER_EC_GENERIC_DB_STORE_FAILED,
                                       "{kebab_case}");
```

```
  }}
  return TALER_MHD_REPLY_JSON_PACK (connection,
            MHD_HTTP_OK,
            GNUNET_JSON_pack_string ("status", "{screaming_snake_case}_OK"));
}}


MHD_RESULT
TAH_{screaming_snake_case}_handler_put (
  struct TAH_RequestHandler *rh,
  struct MHD_Connection *connection,
  void **connection_cls,
  const char *upload_data,
  size_t *upload_data_size,
  const char *const args[])
{{

  struct TALER_AUDITORDB_{camelCase} dc;


  struct GNUNET_JSON_Specification spec[] = {{

    {pl}

    GNUNET_JSON_spec_end ()
  }};


  json_t *json;

  (void) rh;
  (void) connection_cls;
  (void) upload_data;
  (void) upload_data_size;
  {{
    enum GNUNET_GenericReturnValue res;

    res = TALER_MHD_parse_post_json (connection,
                                     connection_cls,
                                     upload_data,
                                     upload_data_size,
                                     &json);
    if (GNUNET_SYSERR == res)
      return MHD_NO;
    if ((GNUNET_NO == res) ||
        (NULL == json))
      return MHD_YES;
    res = TALER_MHD_parse_json_data (connection,
                                     json,
                                     spec);
    if (GNUNET_SYSERR == res)
    {{
      json_decref (json);
      return MHD_NO;                /* hard failure */
    }}
    if (GNUNET_NO == res)
    {{
      json_decref (json);
      return MHD_YES;               /* failure */
    }}
  }}
```

```
  MHD_RESULT res;

  res = process_inconsistency (connection, &dc);
  GNUNET_JSON_parse_free (spec);

  json_decref (json);
  return res;

}}


void
TEAH_{screaming_snake_case}_PUT_init (void)
{{

}}


void
TEAH_{screaming_snake_case}_PUT_done (void)
{{

}}

    """

    return ret

def taler_auditor_httpd_xyz_put_h_new(snake_case, screaming_snake_case, kebab_case,
     camelCase, pl):

    ret = f"""

    {license}

#ifndef SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_PUT_H
#define SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_PUT_H

#include <microhttpd.h>
#include "taler-auditor-httpd.h"

/**
* Initialize subsystem.
*/
void
TEAH_BAD_{screaming_snake_case}_init (void);

/**
* Shut down subsystem.
*/
void
TEAH_BAD_{screaming_snake_case}_done (void);


/**
* Handle a "/{kebab_case}" request.  Parses the JSON, and, if
* successful, checks the signatures and stores the result in the DB.
*
* @param rh context of the handler
* @param connection the MHD connection to handle
* @param[in,out] connection_cls the connection's closure (can be updated)
* @param upload_data upload data
```

```
* @param[in,out] upload_data_size number of bytes (left) in @a upload_data
* @return MHD result code
*/
MHD_RESULT
TAH_{screaming_snake_case}_PUT_handler (struct TAH_RequestHandler *rh,
                                        struct MHD_Connection *
                                        connection,
                                        void **connection_cls,
                                        const char *upload_data,
                                        size_t *upload_data_size,
                                        const char *const args[]);


#endif // SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_PUT_H

     """


     return ret

def taler_auditor_httpd_xyz_get_new(snake_case, screaming_snake_case, kebab_case,
      camelCase, pl):

     ret = f"""

     {license}


#include "platform.h"
#include <gnunet/gnunet_util_lib.h>
#include <gnunet/gnunet_json_lib.h>
#include <jansson.h>
#include <microhttpd.h>
#include <pthread.h>
#include "taler_json_lib.h"
#include "taler_mhd_lib.h"
#include "taler-auditor-httpd.h"
#include "taler-auditor-httpd_{kebab_case}-get.h"

/**
* Add {kebab_case} to the list.
*
* @param[in,out] cls a `json_t *` array to extend
* @param serial_id location of the @a dc in the database
* @param dc struct of inconsistencies
* @return #GNUNET_OK to continue to iterate, #GNUNET_SYSERR to stop iterating
*/
static enum GNUNET_GenericReturnValue
process_{kebab_case} (void *cls,
                     uint64_t serial_id,
                     const struct
                     TALER_AUDITORDB_{camelCase}
                     *dc)
{{
  json_t *list = cls;
  json_t *obj;

  obj = GNUNET_JSON_PACK (

     {pl}

     );
```

```
  GNUNET_break (0 ==
                json_array_append_new (list,
                                       obj));


  return GNUNET_OK;
}}


/**
*
* @param rh context of the handler
* @param connection the MHD connection to handle
* @param[in,out] connection_cls the connection's closure (can be updated)
* @param upload_data upload data
* @param[in,out] upload_data_size number of bytes (left) in @a upload_data
* @return MHD result code
*/
MHD_RESULT
TAH_{screaming_snake_case}_handler_get (struct TAH_RequestHandler *rh,
                                 struct MHD_Connection *
                                 connection,
                                 void **connection_cls,
                                 const char *upload_data,
                                 size_t *upload_data_size,
                                 const char *const args[])
{{
  json_t *ja;
  enum GNUNET_DB_QueryStatus qs;

  (void) rh;
  (void) connection_cls;
  (void) upload_data;
  (void) upload_data_size;
  if (GNUNET_SYSERR ==
      TAH_plugin->preflight (TAH_plugin->cls))
  {{
    GNUNET_break (0);
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_INTERNAL_SERVER_ERROR,
                                       TALER_EC_GENERIC_DB_SETUP_FAILED,
                                       NULL);
  }}
  ja = json_array ();
  GNUNET_break (NULL != ja);

  int64_t limit = -20;
  uint64_t offset;

  TALER_MHD_parse_request_snumber (connection,
                                   "limit",
                                   &limit);

  if (limit < 0)
    offset = INT64_MAX;
  else
    offset = 0;

  TALER_MHD_parse_request_number (connection,
                                  "offset",
                                  &offset);

```

```
  bool return_suppressed = false;

  struct GNUNET_JSON_Specification spec[] = {{
    GNUNET_JSON_spec_bool ("return_suppressed", &return_suppressed),
    GNUNET_JSON_spec_end ()
  }};

  // read the input json
  json_t *json_in;
  {{
    enum GNUNET_GenericReturnValue res;

    res = TALER_MHD_parse_post_json (connection,
                                     connection_cls,
                                     upload_data,
                                     upload_data_size,
                                     &json_in);
    if (GNUNET_SYSERR == res)
      return MHD_NO;
    if ((GNUNET_NO == res) ||
        (NULL == json_in))
      return MHD_YES;
    res = TALER_MHD_parse_json_data (connection,
                                     json_in,
                                     spec);
    if (GNUNET_SYSERR == res)
    {{
      json_decref (json_in);
      return MHD_NO;                 /* hard failure */
    }}
    if (GNUNET_NO == res)
    {{
      json_decref (json_in);
      return MHD_YES;                 /* failure */
    }}
  }}

  qs = TAH_plugin->get_{snake_case} (
    TAH_plugin->cls,
    limit,
    offset,
    return_suppressed,
    &process_{snake_case},
    ja);

  if (0 > qs)
  {{
    GNUNET_break (GNUNET_DB_STATUS_HARD_ERROR == qs);
    json_decref (ja);
    TALER_LOG_WARNING (
      "Failed to handle GET /{kebab_case}\n");
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_INTERNAL_SERVER_ERROR,
                                       TALER_EC_GENERIC_DB_FETCH_FAILED,
                                       "{kebab_case}");
  }}
  return TALER_MHD_REPLY_JSON_PACK (
    connection,
    MHD_HTTP_OK,
    GNUNET_JSON_pack_array_steal ("{kebab_case}",
                                  ja));
}}
```

```


    """

    return ret

def taler_auditor_httpd_xyz_get_h_new(snake_case, screaming_snake_case, kebab_case,
     camelCase, pl):

    ret = f"""

    {license}

    #ifndef SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_GET_H
#define SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_GET_H

#include <gnunet/gnunet_util_lib.h>
#include <microhttpd.h>
#include "taler-auditor-httpd.h"

/**
* Initialize subsystem.
*/
void
TEAH_{screaming_snake_case}_GET_init (void);

/**
* Shut down subsystem.
*/
void
TEAH_BAD_{screaming_snake_case}_GET_done (void);

/**
* Handle a "/{kebab_case}" request.
*
* @param rh context of the handler
* @param connection the MHD connection to handle
* @param[in,out] connection_cls the connection's closure (can be updated)
* @param upload_data upload data
* @param[in,out] upload_data_size number of bytes (left) in @a upload_data
* @return MHD result code
*/
MHD_RESULT
TAH_{screaming_snake_case}_handler_get (struct TAH_RequestHandler *rh,
                                struct MHD_Connection *
                                connection,
                                void **connection_cls,
                                const char *upload_data,
                                size_t *upload_data_size,
                                const char *const args[]);


#endif // SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_GET_H


    """

    return ret

def taler_auditor_httpd_xyz_del_new(snake_case, screaming_snake_case, kebab_case,
     camelCase, pl):

```

```
    ret = f"""

    {license}

#include "taler-auditor-httpd_{kebab_case}-del.h"


MHD_RESULT
TAH_{screaming_snake_case}_handler_delete (struct TAH_RequestHandler *rh,
                                   struct MHD_Connection *
                                   connection,
                                   void **connection_cls,
                                   const char *upload_data,
                                   size_t *upload_data_size,
                                   const char *const args[])
{{

  MHD_RESULT res;
  enum GNUNET_DB_QueryStatus qs;

  uint64_t row_id;

  if (args[1] != NULL)
    row_id = atoi (args[1]);
  else
    return TALER_MHD_reply_with_error (connection,
        MHD_HTTP_BAD_REQUEST,
        // TODO: not the correct ec
        TALER_EC_AUDITOR_DEPOSIT_CONFIRMATION_SIGNATURE_INVALID,
        "exchange signature invalid");

  if (GNUNET_SYSERR ==
      TAH_plugin->preflight (TAH_plugin->cls))
  {{
    GNUNET_break (0);
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_INTERNAL_SERVER_ERROR,
                                       TALER_EC_GENERIC_DB_SETUP_FAILED,
                                       NULL);
  }}


  // execute the transaction
  qs = TAH_plugin->delete_{snake_case} (TAH_plugin->cls,
                                          row_id);

  if (0 == qs)
  {{
    // goes in here if there was an error with the transaction
    GNUNET_break (GNUNET_DB_STATUS_HARD_ERROR == qs);
    TALER_LOG_WARNING (
      "Failed to handle DELETE /{kebab_case}/ %s",
      args[1]);
    return TALER_MHD_reply_with_error (connection,
                   MHD_HTTP_NOT_FOUND,
                   // TODO: not the correct ec
                   TALER_EC_AUDITOR_DEPOSIT_CONFIRMATION_SIGNATURE_INVALID,
                   "exchange signature invalid");

  }}

  // on success?
```

```
  return TALER_MHD_REPLY_JSON_PACK (connection,
                    MHD_HTTP_NO_CONTENT,
                    GNUNET_JSON_pack_string ("status",
                    "{screaming_snake_case}_OK"));

  return res;
}}

    """

    return ret

def taler_auditor_httpd_xyz_del_h_new(snake_case, screaming_snake_case, kebab_case,
     camelCase, pl):

    ret = f"""

    {license}

#ifndef SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_DEL_H
#define SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_DEL_H


#include <microhttpd.h>
#include "taler-auditor-httpd.h"

/**
 * Initialize subsystem.
 */
void
TEAH_{screaming_snake_case}_DELETE_init (void);

/**
 * Shut down subsystem.
 */
void
TEAH_{screaming_snake_case}_DELETE_done (void);

/**
 * Handle a "/{kebab_case}" request.  Parses the JSON, and, if
 * successful, checks the signatures and stores the result in the DB.
 *
 * @param rh context of the handler
 * @param connection the MHD connection to handle
 * @param[in,out] connection_cls the connection's closure (can be updated)
 * @param upload_data upload data
 * @param[in,out] upload_data_size number of bytes (left) in @a upload_data
 * @return MHD result code
  */
MHD_RESULT
TAH_{screaming_snake_case}_handler_delete (struct TAH_RequestHandler *rh,
                                    struct MHD_Connection *
                                    connection,
                                    void **connection_cls,
                                    const char *upload_data,
                                    size_t *upload_data_size,
                                    const char *const args[]);


#endif // SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_DEL_H
```

```
    """

    return ret

def taler_auditor_httpd_xyz_upd_new(snake_case, screaming_snake_case, kebab_case,
     camelCase, pl):

    ret = f"""

    {license}

#include "platform.h"
#include <gnunet/gnunet_util_lib.h>
#include <gnunet/gnunet_json_lib.h>
#include <jansson.h>
#include <microhttpd.h>
#include <pthread.h>
#include "taler_json_lib.h"
#include "taler_mhd_lib.h"
#include "taler-auditor-httpd.h"
#include "taler-auditor-httpd_{kebab_case}-upd.h"

MHD_RESULT
TAH_{screaming_snake_case}_handler_update (
  struct TAH_RequestHandler *rh,
  struct MHD_Connection *connection,
  void **connection_cls,
  const char *upload_data,
  size_t *upload_data_size,
  const char *const args[])
{{
  enum GNUNET_DB_QueryStatus qs;

  if (GNUNET_SYSERR ==
      TAH_plugin->preflight (TAH_plugin->cls))
  {{
    GNUNET_break (0);
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_INTERNAL_SERVER_ERROR,
                                       TALER_EC_GENERIC_DB_SETUP_FAILED,
                                       NULL);
  }}

  struct TALER_AUDITORDB_Generic_Update gu;

  struct GNUNET_JSON_Specification spec[] = {{

    GNUNET_JSON_spec_uint64 ("row_id", &gu.row_id),
    GNUNET_JSON_spec_bool ("suppressed", &gu.suppressed),

    GNUNET_JSON_spec_end ()
  }};

  json_t *json;

  (void) rh;
  (void) connection_cls;
  (void) upload_data;
  (void) upload_data_size;
  {{
    enum GNUNET_GenericReturnValue res;

```

```
    res = TALER_MHD_parse_post_json (connection,
                                     connection_cls,
                                     upload_data,
                                     upload_data_size,
                                     &json);
    if (GNUNET_SYSERR == res)
      return MHD_NO;
    if ((GNUNET_NO == res) ||
        (NULL == json))
      return MHD_YES;
    res = TALER_MHD_parse_json_data (connection,
                                     json,
                                     spec);
    if (GNUNET_SYSERR == res)
    {{
      json_decref (json);
      return MHD_NO;                    /* hard failure */
    }}
    if (GNUNET_NO == res)
    {{
      json_decref (json);
      return MHD_YES;                   /* failure */
    }}
  }}

  /* execute transaction */
  qs = TAH_plugin->update_{snake_case} (TAH_plugin->cls, &gu);

  GNUNET_JSON_parse_free (spec);
  json_decref (json);

  MHD_RESULT ret = MHD_NO;

  switch (qs)
  {{
  case GNUNET_DB_STATUS_HARD_ERROR:
    GNUNET_break (0);
    ret = TALER_MHD_reply_with_error (connection,
                                      MHD_HTTP_INTERNAL_SERVER_ERROR,
                                      TALER_EC_GENERIC_DB_STORE_FAILED,
                                      "update_account");
    break;
  case GNUNET_DB_STATUS_SOFT_ERROR:
    GNUNET_break (0);
    ret = TALER_MHD_reply_with_error (connection,
                                      MHD_HTTP_INTERNAL_SERVER_ERROR,
                                      TALER_EC_GENERIC_INTERNAL_INVARIANT_FAILURE,
                                      "unexpected serialization problem");
    break;
  case GNUNET_DB_STATUS_SUCCESS_NO_RESULTS:
    return TALER_MHD_reply_with_error (connection,
                                       MHD_HTTP_NOT_FOUND,
                                       TALER_EC_MERCHANT_GENERIC_ACCOUNT_UNKNOWN,
                                       "no updates executed");
    break;
  case GNUNET_DB_STATUS_SUCCESS_ONE_RESULT:
    ret = TALER_MHD_reply_static (connection,
                                  MHD_HTTP_NO_CONTENT,
                                  NULL,
                                  NULL,
                                  0);
    break;
```

```
  }}

  return ret;
}}

    """

    return ret

def taler_auditor_httpd_xyz_upd_h_new(snake_case, screaming_snake_case, kebab_case, camelCase, pl):
    ret = f"""

    {license}

#ifndef SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_UPD_H
#define SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_UPD_H


#include <microhttpd.h>
#include "taler-auditor-httpd.h"

MHD_RESULT
TAH_{screaming_snake_case}_handler_update (struct TAH_RequestHandler *rh,
                                    struct MHD_Connection *
                                    connection,
                                    void **connection_cls,
                                    const char *upload_data,
                                    size_t *upload_data_size,
                                    const char *const args[]);

#endif // SRC_TALER_AUDITOR_HTTPD_{screaming_snake_case}_UPD_H


    """
    return ret

def httpd(words, comp):

    pl = ""
    for w in words.items():
        pl += spjs(w[0], w[1]) + ",\n"

    sc = "_".join(comp)
    ssc = "_".join(map(str.upper,comp))
    kc = "-".join(comp)
    cc = "".join(map(str.capitalize,comp))

    p = taler_auditor_httpd_xyz_put_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-put.c","w+")
    f.write(p)
    f.close()

    p = taler_auditor_httpd_xyz_put_h_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-put.h","w+")
    f.write(p)
    f.close()

    pl = ""
    for w in words.items():
```

```
        pl += pkjs(w[0], w[1]) + ",\n"

    p = taler_auditor_httpd_xyz_get_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-get.c","w+")
    f.write(p)
    f.close()

    p = taler_auditor_httpd_xyz_get_h_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-get.h","w+")
    f.write(p)
    f.close()

    p = taler_auditor_httpd_xyz_del_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-del.c","w+")
    f.write(p)
    f.close()

    p = taler_auditor_httpd_xyz_del_h_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-del.h","w+")
    f.write(p)
    f.close()

    p = taler_auditor_httpd_xyz_upd_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-upd.c","w+")
    f.write(p)
    f.close()

    p = taler_auditor_httpd_xyz_upd_h_new(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditor/taler-auditor-httpd_" + kc + "-upd.h","w+")
    f.write(p)
    f.close()



def pg_del(snake_case, screaming_snake_case, kebab_case, camelCase, pl):

    ret = f"""

    {license}

#include "pg_del_{snake_case}.h"

#include "taler_pq_lib.h"
#include "pg_helper.h"

enum GNUNET_DB_QueryStatus
TAH_PG_del_{snake_case} (
  void *cls,
  uint64_t row_id)
{{
  struct PostgresClosure *pg = cls;
  struct GNUNET_PQ_QueryParam params[] = {{
    GNUNET_PQ_query_param_uint64 (&row_id),
    GNUNET_PQ_query_param_end
  }};

```

```
  PREPARE (pg,
            "auditor_delete_{snake_case}",
            "DELETE"
            " FROM auditor_{snake_case}"
            " WHERE row_id=$1;");
  return GNUNET_PQ_eval_prepared_non_select (pg->conn,
                                             "auditor_delete_{snake_case}",
                                             params);
}}

    """

    return ret

def pg_del_h(snake_case, screaming_snake_case, kebab_case, camelCase, pl):
    ret = f"""

    {license}

#ifndef SRC_PG_DEL_{screaming_snake_case}_H
#define SRC_PG_DEL_{screaming_snake_case}_H

#include "taler_util.h"
#include "taler_auditordb_plugin.h"

/**
 * Delete a row from the bad sig losses table.
 *
 * @param cls the @e cls of this struct with the plugin-specific state
 * @param row_id row to delete
 * @return query transaction status
 */
enum GNUNET_DB_QueryStatus
TAH_PG_del_{snake_case} (
  void *cls,
  uint64_t row_id);

#endif // SRC_PG_DEL_{screaming_snake_case}_H


    """

    return ret

def pg_upd(snake_case, screaming_snake_case, kebab_case, camelCase, pl):

    ret = f"""

{license}

#include "platform.h"
#include "taler_pq_lib.h"
#include "pg_helper.h"

#include "pg_update_{snake_case}.h"

/*
Update a given resource  for now this only means suppressing
*/
enum GNUNET_DB_QueryStatus
TAH_PG_update_{snake_case} (
  void *cls,
```

```
  const struct TALER_AUDITORDB_Generic_Update *gu)
{{
  struct PostgresClosure *pg = cls;
  struct GNUNET_PQ_QueryParam params[] = {{
    GNUNET_PQ_query_param_uint64 (&gu->row_id),
    GNUNET_PQ_query_param_bool (gu->suppressed),
    GNUNET_PQ_query_param_end
  }};


  PREPARE (pg,
           "update_{snake_case}",
           "UPDATE auditor_{snake_case} SET"
           " suppressed=$2"
           " WHERE row_id=$1");
  return GNUNET_PQ_eval_prepared_non_select (pg->conn,
                                             "update_{snake_case}",
                                             params);
}}


    """

    return ret

def pg_upd_h(snake_case, screaming_snake_case, kebab_case, camelCase, pl):

    ret = f"""

    {license}

#ifndef SRC_PG_UPDATE_{screaming_snake_case}_H
#define SRC_PG_UPDATE_{screaming_snake_case}_H

#include "taler_util.h"
#include "taler_auditordb_plugin.h"

enum GNUNET_DB_QueryStatus
TAH_PG_update_{snake_case} (
  void *cls,
  const struct TALER_AUDITORDB_Generic_Update *dc);

#endif // SRC_PG_UPDATE_{screaming_snake_case}_H


    """

    return ret

def pg_insert(snake_case, screaming_snake_case, kebab_case, camelCase, pl, sql_i):

    ret = f"""

    {license}

#include "platform.h"
#include "taler_pq_lib.h"
#include "pg_helper.h"

#include "pg_insert_{snake_case}.h"

enum GNUNET_DB_QueryStatus
```

```
TAH_PG_insert_{snake_case} (
  void *cls,
  const struct TALER_AUDITORDB_{camelCase} *dc)
{{
  struct PostgresClosure *pg = cls;
  struct GNUNET_PQ_QueryParam params[] = {{

    {pl}

    GNUNET_PQ_query_param_end
  }};

  PREPARE (pg,
           "auditor_{snake_case}_insert",
           "INSERT INTO auditor_{snake_case} "
           {sql_i}
           );
  return GNUNET_PQ_eval_prepared_non_select (pg->conn,
                                             "auditor_{snake_case}_insert",
                                             params);
}}

    """

    return ret

def pg_insert_h(snake_case, screaming_snake_case, kebab_case, camelCase, pl, sql_i):

    ret = f"""

    {license}



#ifndef SRC_PG_INSERT_{screaming_snake_case}_H
#define SRC_PG_INSERT_{screaming_snake_case}_H

#include "taler_util.h"
#include "taler_auditordb_plugin.h"


/**
 * Insert information about a bad sig loss into the database.
 *
 * @param cls the @e cls of this struct with the plugin-specific state
 * @param dc deposit confirmation information to store
 * @return query result status
 */
enum GNUNET_DB_QueryStatus
TAH_PG_insert_{snake_case} (
  void *cls,
  const struct TALER_AUDITORDB_{camelCase} *dc);

#endif // SRC_PG_INSERT_{screaming_snake_case}_H


    """

    return ret

def pg_get(snake_case, screaming_snake_case, kebab_case, camelCase, pl, sql_i):

```

```
    ret = f"""

{license}

#include "platform.h"
#include "taler_error_codes.h"
#include "taler_dbevents.h"
#include "taler_pq_lib.h"
#include "pg_helper.h"

#include "pg_get_{snake_case}.h"


struct {camelCase}Context
{{

  /**
   * Function to call for each bad sig loss.
   */
  TALER_AUDITORDB_{camelCase}Callback cb;

  /**
   * Closure for @e cb
   */
  void *cb_cls;

  /**
   * Plugin context.
   */
  struct PostgresClosure *pg;

  /**
   * Query status to return.
   */
  enum GNUNET_DB_QueryStatus qs;
}};


/**
 * Helper function for #TAH_PG_get_{snake_case}().
 * To be called with the results of a SELECT statement
 * that has returned @a num_results results.
 *
 * @param cls closure of type `struct {camelCase}Context *`
 * @param result the postgres result
 * @param num_results the number of results in @a result
 */
static void
{snake_case}_cb (void *cls,
                    PGresult *result,
                    unsigned int num_results)
{{
  struct {camelCase}Context *dcc = cls;
  struct PostgresClosure *pg = dcc->pg;

  for (unsigned int i = 0; i < num_results; i++)
  {{
    uint64_t serial_id;

    struct TALER_AUDITORDB_{camelCase} dc;

    struct GNUNET_PQ_ResultSpec rs[] = {{
```

```

      GNUNET_PQ_result_spec_uint64 ("row_id", &serial_id),

      {pl}

      GNUNET_PQ_result_spec_end
    }};
    enum GNUNET_GenericReturnValue rval;

    if (GNUNET_OK !=
        GNUNET_PQ_extract_result (result,
                                  rs,
                                  i))
    {{
      GNUNET_break (0);
      dcc->qs = GNUNET_DB_STATUS_HARD_ERROR;
      return;
    }}

    dcc->qs = i + 1;

    rval = dcc->cb (dcc->cb_cls,
                    serial_id,
                    &dc);
    GNUNET_PQ_cleanup_result (rs);
    if (GNUNET_OK != rval)
      break;
  }}
}}


enum GNUNET_DB_QueryStatus
TAH_PG_get_{snake_case} (
  void *cls,
  int64_t limit,
  uint64_t offset,
  bool return_suppressed,            // maybe not needed
  TALER_AUDITORDB_{camelCase}Callback cb,
  void *cb_cls)
{{

  struct PostgresClosure *pg = cls;
  struct GNUNET_PQ_QueryParam params[] = {{
    GNUNET_PQ_query_param_uint64 (&offset),
    GNUNET_PQ_query_param_bool (return_suppressed),
    GNUNET_PQ_query_param_int64 (&limit),
    GNUNET_PQ_query_param_end
  }};
  struct {camelCase}Context dcc = {{
    .cb = cb,
    .cb_cls = cb_cls,
    .pg = pg
  }};
  enum GNUNET_DB_QueryStatus qs;

  PREPARE (pg,
           "auditor_{snake_case}_get_desc",
           "SELECT"
           {sql_i}
           " FROM auditor_{snake_case}"
           " WHERE (row_id < $1)"
           " AND ($2 OR suppressed is false)"
```

```
            " ORDER BY row_id DESC"
            "  LIMIT $3"
            );
  PREPARE (pg,
            "auditor_{snake_case}_get_asc",
            "SELECT"
            {sql_i}
            " FROM auditor_{snake_case}"
            " WHERE (row_id > $1)"
            " AND ($2 OR suppressed is false)"
            " ORDER BY row_id ASC"
            "  LIMIT $3"
            );
  qs = GNUNET_PQ_eval_prepared_multi_select (pg->conn,
                                             (limit > 0)
                                             ? "auditor_{snake_case}_get_asc"
                                             : "auditor_{snake_case}_get_desc",
                                             params,
                                             &{snake_case}_cb,
                                             &dcc);

  if (qs > 0)
    return dcc.qs;
  GNUNET_break (GNUNET_DB_STATUS_HARD_ERROR != qs);
  return qs;
}}


    """

    return ret

def pg_get_h(snake_case, screaming_snake_case, kebab_case, camelCase, pl, sql_i):

    ret = f"""

    {license}

#ifndef SRC_PG_GET_{screaming_snake_case}_H
#define SRC_PG_GET_{screaming_snake_case}_H

#include "taler_util.h"
#include "taler_json_lib.h"
#include "taler_auditordb_plugin.h"

/**
 * Get information about {kebab_case} from the database.
 *
 * @param cls the @e cls of this struct with the plugin-specific state
 * @param start_id row/serial ID where to start the iteration (0 from
 *                 the start, exclusive, i.e. serial_ids must start from 1)
 * @param return_suppressed should suppressed rows be returned anyway?
 * @param cb function to call with results
 * @param cb_cls closure for @a cb
 * @return query result status
 */
enum GNUNET_DB_QueryStatus
TAH_PG_get_{snake_case} (
  void *cls,
  int64_t limit,
  uint64_t offset,
  bool return_suppressed,
```

```
  TALER_AUDITORDB_{camelCase}Callback cb,
  void *cb_cls);

#endif // SRC_PG_GET_{screaming_snake_case}_H


    """

    return ret


def pg_auditor(words, comp):

    pl = ""
    for w in words.items():
        pl += qupq(w[0], w[1]) + ",\n"


    sql_i = ""
    sql_c = 0
    sql_a = ""
    for w in words.items():
        if (sql_c == 0):
            sql_i += "\"(␣" + w[0] + ",\"\n"
        else:
            sql_i += "\"␣" + w[0] + ",\"\n"
        sql_c += 1
        sql_a += f"${sql_c},"

    sql_i = sql_i.removesuffix(",\"\n") + "\"\n"
    sql_a = sql_a.removesuffix(",")
    sql_i += f"\")␣VALUES␣({sql_a});\""


    sc = "_".join(comp)
    ssc = "_".join(map(str.upper,comp))
    kc = "-".join(comp)
    cc = "".join(map(str.capitalize,comp))


    pl_b = ""
    for w in words.items():
        if (w[0] == "row_id"):
            continue
        pl_b += qupq(w[0], w[1]) + ",\n"

    p = pg_insert(sc,ssc,kc,cc,pl_b,sql_i)

    f = open("taler-files/auditordb/pg_insert_" + sc + ".c","w+")
    f.write(p)
    f.close()

    p = pg_insert_h(sc,ssc,kc,cc,pl_b,sql_i)

    f = open("taler-files/auditordb/pg_insert_" + sc + ".h","w+")
    f.write(p)
    f.close()


    sql_i = ""
    for w in words.items():
        sql_i += "\"␣" + w[0] + ",\"\n"
```

```
    sql_i = sql_i.removesuffix(",\"\n") + "\""


    sc = "_".join(comp)
    ssc = "_".join(map(str.upper,comp))
    kc = "-".join(comp)
    cc = "".join(map(str.capitalize,comp))

    pl_c = ""
    for w in words.items():
        if (w[0] == "row_id"):
            continue
        pl_c += sppq(w[0], w[1]) + ",\n"

    p = pg_get(sc,ssc,kc,cc,pl_c,sql_i)

    f = open("taler-files/auditordb/pg_get_" + sc + ".c","w+")
    f.write(p)
    f.close()

    p = pg_get_h(sc,ssc,kc,cc,pl_c,sql_i)

    f = open("taler-files/auditordb/pg_get_" + sc + ".h","w+")
    f.write(p)
    f.close()



    p = pg_upd(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditordb/pg_update_" + sc + ".c","w+")
    f.write(p)
    f.close()

    p = pg_upd_h(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditordb/pg_update_" + sc + ".h","w+")
    f.write(p)
    f.close()



    p = pg_del(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditordb/pg_del_" + sc + ".c","w+")
    f.write(p)
    f.close()

    p = pg_del_h(sc,ssc,kc,cc,pl)

    f = open("taler-files/auditordb/pg_del_" + sc + ".h","w+")
    f.write(p)
    f.close()


def taler_auditor_httpd(amalgamation):
      print("taler-auditor-httpd")

      for a in amalgamation:

        sc = a[2]
```

```
        ssc = a[3]
        kc = a[4]
        cc = a[5]

        print(f"""
                #include "taler-auditor-httpd_{kc}-del.h"
                #include "taler-auditor-httpd_{kc}-put.h"
                #include "taler-auditor-httpd_{kc}-get.h"
                #include "taler-auditor-httpd_{kc}-upd.h"
              """)

def plugin_auditordb_postgres(amalgamation):

    print("plugin_auditordb_postgres:\n")

    for a in amalgamation:

      sc = a[2]
      ssc = a[3]
      kc = a[4]
      cc = a[5]



      print(f"""
              #include "pg_get_{sc}.h"
              #include "pg_del_{sc}.h"
              #include "pg_insert_{sc}.h"
              #include "pg_update_{sc}.h"
            """)

      print(f"""
              plugin->delete_{sc} = &TAH_PG_del_{sc};
              plugin->insert_{sc} = &TAH_PG_insert_{sc};
              plugin->get_{sc} = &TAH_PG_get_{sc};
              plugin->update_{sc} = &TAH_PG_update_{sc};
            """)

def taler_auditordb_plugin(amalgamation):

    print("taler_auditordb_plugin.h:\n")

    for a in amalgamation:

      sc = a[2]
      ssc = a[3]
      kc = a[4]
      cc = a[5]

      words = a[0]



      str_content = ""
      for w in words.items():
          if w[0] == "row_id":
              continue
          str_content += c_types[w[1]] + " " + w[0] + ";\n"

      print(f"""
  struct TALER_AUDITORDB_{cc}
  {{
```

```
  unsigned int row_id;
  {str_content}
  }};
          """)

      print(f"""
  typedef enum GNUNET_GenericReturnValue
  (*TALER_AUDITORDB_{cc}Callback)(
  void *cls,
  uint64_t serial_id,
  const struct TALER_AUDITORDB_{cc} *dc);
  """)

      print(f"""
  enum GNUNET_DB_QueryStatus
  (*get_{sc})(
  void *cls,
  int64_t limit,
  uint64_t offset,
  bool return_suppressed,
  TALER_AUDITORDB_{cc}Callback cb,
  void *cb_cls);
          """)

      print(f"""
  enum GNUNET_DB_QueryStatus
  (*delete_{sc})(
  void *cls,
  uint64_t row_id);
          """)

      print(f"""
  enum GNUNET_DB_QueryStatus
  (*insert_{sc})(
  void *cls,
  const struct TALER_AUDITORDB_{cc} *dc);
          """)


      print(f"""
  enum GNUNET_DB_QueryStatus
  (*update_{sc})(
  void *cls,
  const struct TALER_AUDITORDB_Generic_Update *gu);
          """)

def makefile_auditordb(amalgamation):


  print("\nmakefile auditordb\n")

  for a in amalgamation:

    sc = a[2]
    ssc = a[3]
    kc = a[4]
    cc = a[5]

    print(f"""
pg_get_{sc}.c pg_get_{sc}.h \\
pg_del_{sc}.c pg_del_{sc}.h \\
pg_insert_{sc}.c pg_insert_{sc}.h \\
```

```
pg_update_{sc}.c pg_update_{sc}.h \\
""")

def makefile_auditor(amalgamation):


  print("\nmakefile␣auditor\n")

  for a in amalgamation:

    sc = a[2]
    ssc = a[3]
    kc = a[4]
    cc = a[5]




    print(f"""
taler-auditor-httpd_{kc}-del.c taler-auditor-httpd_{kc}-del.h \\
taler-auditor-httpd_{kc}-put.c taler-auditor-httpd_{kc}-put.h \\
taler-auditor-httpd_{kc}-get.c taler-auditor-httpd_{kc}-get.h \\
taler-auditor-httpd_{kc}-upd.c taler-auditor-httpd_{kc}-upd.h \\
""")


def taler_auditor_httpd_again(amalgamation):


  print("\ntaler-auditor-httpd\n")

  for a in amalgamation:

    sc = a[2]
    ssc = a[3]
    kc = a[4]
    cc = a[5]



    print(f"""
{{ "/{kc}", MHD_HTTP_METHOD_GET,
  "application/json",
  NULL, 0,
  &TAH_{ssc}_handler_get,
  MHD_HTTP_OK }},
{{ "/{kc}", MHD_HTTP_METHOD_PUT,
  "application/json",
  NULL, 0,
  &TAH_{ssc}_handler_put,
  MHD_HTTP_OK }},
{{ "/{kc}", MHD_HTTP_METHOD_DELETE,
  "application/json",
  NULL, 0,
  &TAH_{ssc}_handler_delete,
  MHD_HTTP_OK }},
{{ "/{kc}", MHD_HTTP_METHOD_PATCH,
  "application/json",
  NULL, 0,
  &TAH_{ssc}_handler_update,
  MHD_HTTP_OK }},
  """)
```

```python
def main():

  amalgamation = list()

  directory = os.fsencode("taler-files/sql")

  for file in os.listdir(directory):

      words = {}

      name = os.fsdecode(file)
      path = os.fsdecode(directory)

      if (name.find("DS-Store")):
        continue

      nm = name.removesuffix(".sql")
      comp = list( filter(lambda x: x != "0002-auditor",nm.split('_')) )


      sql = open(path + '/' +  name, 'r', encoding='utf-8', errors='ignore')

      lines = sql.readlines()

      i = 0
      for line in lines:
          #find point of interest
          if (line.find("CREATE TABLE") < 0):
              i += 1
              continue
          else:
              i += 1
              # skips one, but that is ok
              exit = 0
              for x in range(i,len(lines) - 1):
                  sql = lines[x]

                  if (sql.find(");") >= 0):
                      exit = 1

                  if (exit == 0):
                      sql = re.sub(r'[^\w\s]', '', sql)

                      if (sql != '\n'):

                          dingdong = sql.split(' ')

                          bloop = list(filter(lambda x: x != '',dingdong))

                          #print(bloop)
                          words[bloop[0].strip().lower()] = bloop[1].strip().lower()



      httpd(words, comp)

      pg_auditor(words,comp)

      # copy paste

      sc = "_".join(comp)
```

```
        ssc = "_".join(map(str.upper,comp))
        kc = "-".join(comp)
        cc = "".join(map(str.capitalize,comp))

        tpl = (words, comp, sc, ssc, kc, cc)

        amalgamation.append(tpl)



    taler_auditor_httpd(amalgamation)

    plugin_auditordb_postgres(amalgamation)

    taler_auditordb_plugin(amalgamation)

    makefile_auditordb(amalgamation)


    makefile_auditor(amalgamation)

    taler_auditor_httpd_again(amalgamation)


if __name__ == "__main__":
    main()
```