Privacy Preserving Subscription and Discounts

Designing and implementing token-based subscriptions
and discounts in GNU Taler

Christian Blättler
christian.blaettler@taler.net

June 13, 2024

A thesis presented for the degree of
Bachelor of Science

Advisors	Prof. Dr. Christian Grothoff	Bern University of Applied Sciences
	Prof. Dr. Emmanuel Benoist	School of Engineering & Computer Science
Expert	Han van der Kleij	Computer Science

Abstract

Subscription-based services are more popular than ever, with a significant portion of digital goods such as newspaper articles, music, movies, and TV shows sold through this business model. These services are generally tied to a user account and as a result, subscription usage leaves a data trail. Service providers can use the collected usage data to build a personality profile that can reveal information about sensitive topics such as political views, sexual orientation, and health complications.

This information, in the wrong hands, can have critical implications, especially in regions with repressive regimes. Ideally, a solution to this problem also addresses the challenge of subscribers sharing their credentials with groups of people online.

This thesis presents a solution for account-less and privacy-preserving subscriptions based on GNU Taler. The solution is convenient for customers, affordable for merchants, and resistant to abusive sharing of subscriptions. Subscriptions are valid during a configured timeframe, while usage during that timeframe is unlimited. The flexible design of the solution allows it to be used for a wide array of use cases beyond subscriptions, such as discounts, loyalty stamps, multi-entry event ticketing, membership programs, deposit systems, and privacy-preserving gifts. In addition, the solution's low operational costs, coupled with its built-in protection against abusive sharing of subscriptions, make it highly attractive to merchants.

The solution is implemented in the GNU Taler merchant component as free and open source software. The integration into the GNU Taler wallets is subject to future work. Furthermore, to inform customers about the degree of anonymity for a given subscription, an additional service for authorizing the anonymity set size of subscriptions is proposed.

Acknowledgements

I would like to thank my advisors, Prof. Dr. Christian Grothoff and Prof. Dr. Emmanuel Benoist, for their guidance, continuous feedback, and prompt responses to any questions that arose during the writing of my thesis. Their ideas and mentorship were instrumental in shaping and refining my work. The design document in Appendix B served as the initial basis for my thesis and I'd like to thank Prof. Dr. Christian Grothoff and Florian Dold for authoring it. I would also like to thank the GNU Taler community for the contributions to the API documentation included in Appendix A.

I would also like to acknowledge the previous work conducted in the 'Project 2' module, which served as a preparation for my thesis. The resulting output was a preliminary version of the token family management API and the accompanying merchant back-office user interface, which were extended and documented during the writing of this thesis.

Additionally, I am grateful for the assistance provided by general-purpose large language models, such as ChatGPT[1] and Claude[2], in helping me refine and improve the clarity and flow of my writing throughout this process.

Finally, I would like to express my sincere gratitude to all those who supported me during the writing of this thesis, be it by proofreading or providing input on technical challenges.

[1] `https://chatgpt.com/`
[2] `https://claude.ai/`

Contents

List of Figures

List of Tables

Acronyms

SPA	Single Page Application. 27, 45, 49, 50
POS	Point of Sale. 11, 27
DOM	Document Object Model. 45
GUI	Graphical User Interface. 45
SWR	Stale-While-Revalidate. 50
OCR	Optical Character Recognition. 28

List of Symbols

k_{issue} Token issue private key used to issue tokens.

K_{issue} Token issue public key used to verify tokens.

$(k_{\text{issue}}, K_{\text{issue}})$ Token issue key pair generated by the merchant.

s'_{issue} Blinded token issue signature made by the merchant with k_{issue} on h'_{use}.

s_{issue} Token issue signature unblinded from s'_{issue}.

k_{use} Token use private key used to confirm usage of tokens.

K_{use} Token use public key used to verify usage of tokens.

$(k_{\text{use}}, K_{\text{use}})$ Token use key pair generated by the wallet.

h_{use} Hash of token use public key K_{use}.

h'_{use} Blinded token use public key hash h_{use}.

s_{use} Token use signature made by the wallet with k_{use}.

Sign_{use} Function to sign a token use request m using private key k. Prepends the signature header to m before signing. Header consists of the purpose identifier (defined in Table 3.1) and the length of the signed data.
Parameters: private key k, token use request m.

Verify Function to verify a signature. Stops protocol execution and returns error upon encountering an invalid signature.
Parameters: public key K, message m, signature s.

$\text{Verify}_{\text{use}}$ Function to verify a signature done by Sign_{use}. Stops protocol execution and returns error upon encountering an invalid signature.
Parameters: public key K, token use request m, token use signature s_{use}.

c Contract JSON object.

h_c Hash of normalized contract JSON object c.

w Wallet data JSON object containing output token envelopes and index of selected choice.

h_w Hash of normalized wallet data JSON object w.

b Blinding secret.

BlindSecretGen Function to generate a blinding secret b.

Blind Function to blind a message m for public key K with blinding secret b.
Parameters: public key K, message m, blinding secret b.

Unblind Function to unblind a message m' previously blinded by Blind(K, m, b).
Parameters: public key K, blinded message m', blinding secret b.

Hash Cryptographic hash function.
Parameters: message m.

KeyGen Generate keypair (k, K) consisting of private key k and public key K.

1. Introduction

In the digital age, subscription-based business models have become the dominant mode of consumption for digital goods, including music, movies, TV shows, newspapers, magazines, and software. These models provide a convenient user experience that allows users to access vast libraries of content. However, this convenience comes at a significant cost to user privacy. This thesis proposes a solution for paid subscriptions that avoids the inherent violation of customer privacy. The proposed solution employs modern cryptography to allow customers to use their subscriptions without registering for an account. This approach enables anonymous, unlimited, and unlinkable access during a specified validity period while resisting a multitude of potential abuses. The solution can be adapted to accommodate a diverse range of use cases, including discounts.

1.1. Motivation

In order to access conventional subscription-based services, customers are required to create accounts, thereby linking their consumption habits to their identities. Every interaction is recorded, including the articles read and the songs played, which are then fed into a larger profile constructed by the service providers. Furthermore, payment processors such as Mastercard, Visa, and PayPal add another layer of tracking[1]. Visa plans to be not only your payment provider, but also your identity provider [2].

The vast amounts of personal data collected can be used by providers to extract sensitive information about their customers [1]. This may include information on political views, sexual orientation [2], or health problems. Such information, in the wrong hands, can have critical implications, especially in regions with repressive regimes.

Additionally, this information can be used to sell targeted advertising and drive service optimizations aimed at maximizing user engagement. Consequently, the subscription revenue model, which appears to be a direct source of revenue, is in

[1]Visa claims to analyze more than 500 attributes in every transaction: `https://usa.visa.com/run-your-business/visa-security/risk-solutions/authorization-optimization.html`

[2]`https://www.visa.com.vn/en_VN/about-visa/newsroom/press-releases/visa-reinvents-the-card-unveils-new-products-for-digital-age.html`

fact supplemented by a more insidious monetization of user data. This model not only undermines privacy but also manipulates consumption patterns, leveraging data to keep users hooked.

Conventional subscription services store subscriptions in a centralized database, as apparent in Figure 1.1. To use a subscription, the customer must authenticate, thus enabling the merchant to check their database for a matching subscription. The actions of a user within such a system are inherently linkable, and users do not maintain control of their own data, which is a violation of informational self-determination. The customer therefore simply has to trust the merchant that the subscription in the centralized database is not lost or accessed by an unauthorized party.

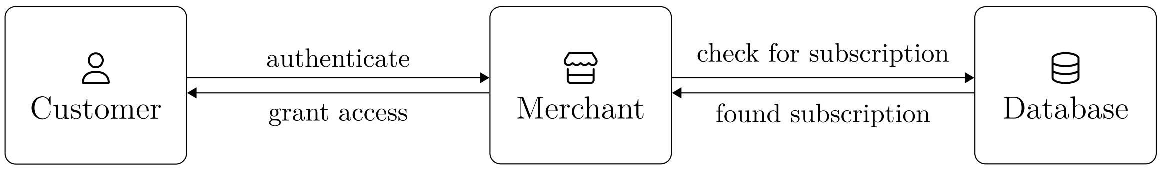


Figure 1.1: Schematic flow of using conventional subscription-based services.

Recognizing these challenges, this thesis proposes a solution to subscriptions and discounts based on blind signatures as originally described by Chaum [3]. This solution prioritizes user privacy by ensuring unlinkability between actions of users. Furthermore, it addresses the problem of abusive sharing of subscriptions, which many merchants struggle to solve.

1.2. Design goals

The objective of this thesis is to design and implement a subscription and discount solution that adheres to the principles of privacy by design. The solution should possess the following characteristics.

1. **Account-less.** The system permits customers to purchase and utilize subscriptions and discounts without the necessity of creating a user account, thereby simplifying the process and enhancing user privacy.
2. **Anonymity.** The system ensures customer anonymity among all subscribers accessing the same type of subscription. Specifically, the system is designed such that customers are not required to disclose any personally identifiable information, such as credit card details, thereby preserving their anonymity.
3. **Unlinkability.** The system ensures that individual customer actions remain unlinkable. This prevents the potential aggregation of customer activities into a comprehensive profile, thereby safeguarding user privacy.
4. **Abuse resistance.** The system incorporates mechanisms that prevent abusive sharing of subscriptions among large groups of people. This property

aligns with the interests of merchants in preventing misuse, thereby fostering merchant adoption.

1.3. Approach

The proposed solution utilizes a wallet storing digitally signed subscription tokens on the customer's device. A wallet can be a mobile app or integrated into a web browser as an extension. Such an extension allows for a seamless user experience using a token-based subscription in a web browser.

Figure 1.2 provides a schematic overview of a token-based subscription in use. In contrast to the conventional subscription flow (Figure 1.1), which involves storing subscriptions in a centralized database, the token-based approach decentralizes the storage of subscriptions on the customer's devices.

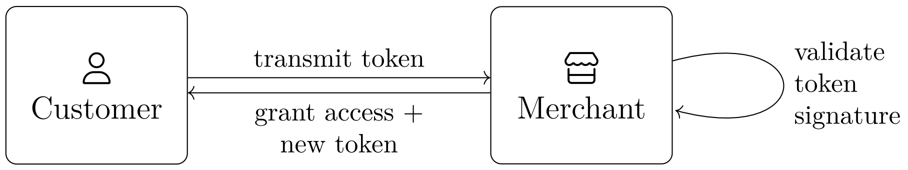


Figure 1.2: Schematic overview of using token-based subscriptions.

While the following sections focus on subscriptions, it should be noted that a slightly modified version of this protocol also allows for use cases such as discounts or loyalty stamps.

Account-less

Subscribers are not required to register with a merchant to purchase a subscription. Upon purchasing a subscription, the merchant issues a blindly signed token to the customer, who then stores it in their digital wallet. This token is a cryptographic proof of subscription. The customer can subsequently use this token to access a subscription good, such as a magazine article or a movie. Upon receiving a token, the merchant verifies its validity date and signature. If a token is valid, the merchant grants the subscriber access. The merchant is responsible for maintaining a record of all tokens that have been previously used, in order to prevent double-spending.

Anonymity

Since tokens are signed blindly, merchants only get to see the blinded version of a token during issuance. Once a customer wishes to use their subscriptions, they transmit the unblinded token to the merchant. This is the first time the merchant sees this unblinded token. The merchant therefore does not have any information about the date of issuance. The only information the merchant can gather from this process is that the customer is an active subscriber.

Unlinkability

Each token can only be used once to ensure unlinkability. When a token is used the merchant issues a fresh, blindly signed token to the customer in the same transaction. This process forms a chain of subscription tokens where only the most recent token is valid.

Misuse prevention

The continuous rotation of subscription tokens also offers the advantage of misuse prevention. Consider a scenario in which an individual shares their subscription token publicly on the internet for other people to use. The first person using that token will receive a new token, rendering the publicly disclosed token invalid and essentially stealing the subscription from the original subscriber. This is in stark contrast to traditional subscriptions: Credentials[3] can be shared with large groups without concern of losing a subscription, since the theft of a traditional subscription typically necessitates access to the associated email account.

Compared to traditional subscriptions, token-based subscriptions require a higher degree of trust among all participants in the sharing group. To successfully share a token-based subscription, a group of honest participants that trust each other is required. In practice, this could be a family or a group of close friends.

1.4. Use cases

The proposed solution provides an efficient framework for addressing a wide range of use cases. In essence, a blindly signed token represents an anonymous voucher redeemable at a specific merchant within a designated validity period. The system's flexibility is significantly enhanced by the ability to generate new tokens through transactions. The following section will examine a number of scenarios in which tokens offer a practical solution and address prevalent privacy and security concerns.

Subscriptions

Subscription tokens are single-use meaning when a subscription token is used, the merchant will issue a new token to the customer. The newly issued token retains the same validity period as the previous one, thus representing the same subscription period.

Token-based subscriptions are not automatically renewed by design. Once expired, the customer must take the initiative to renew their subscription. Since the wallet is aware that the subscription has expired, the process of renewing a subscription can be designed in a frictionless manner. The explicit consent of the customer is still required, which eliminates hard-to-cancel auto-renewing subscriptions [4].

[3]traditionally username and password

Discounts and coupons

Traditional discounts and coupons are often used for user behavior tracking. Utilizing anonymous tokens in this context allows customers to enjoy discounts without compromising their privacy. As an example: a merchant might offer a 20% discount to customers who present a valid discount token. When using such a discount token, it is unclear to the merchant where this token originated from.

Loyalty stamps

Many merchants want to encourage customers to shop at their store on a regular basis. One of the ways they do this is through loyalty programs[4] [5] where users receive discounts or even free items after spending a certain amount. These loyalty systems can also be used to aggregate the purchase history of customers, allowing merchants to profile the consumption habits of their customers. This practice can lead to the collection of sensitive information which in turn raises concerns about privacy.

The proposed solution can be used to build anonymous loyalty programs. Depending on the purchase, merchants can issue a certain number of blindly signed loyalty tokens which essentially act as stamps. The customer can use these tokens on a future purchase to get a discount. For example, restaurants frequently employ a loyalty stamps program, wherein the customer receives their 11th lunch menu at no additional cost. Each time a customer purchases a lunch menu, they receive a blindly signed loyalty token. When purchasing the menu, the customer has the option of paying with traditional payment methods or utilizing ten loyalty tokens. This solution meets the merchant's desire to encourage customers to shop at the store on a regular basis without violating the customer's privacy.

Membership programs

Typically, members of a group possess some form of identification, such as a card or badge, that entitles them to certain privileges. However, in most cases it is sufficient to be aware of one's membership status, rather than the specific identity of the member. Such memberships can be modeled using tokens analogous to subscriptions. Upon signing up, the member receives a member token. This token is exchanged for a fresh token each time it is used. This guarantees the anonymity of the member[6] and keeps their actions unlinkable. Examples of this type of membership program include non-profit organizations and locations that are exclusive to members of a club.

[4]`https://cumulus.migros.ch`

[5]`https://www.supercard.ch/`

[6]within the group of members

Ticketing

A multi-entry ticket to an event can be modeled using tokens. When the ticket is purchased, the customer receives an entry token that is valid for the duration of the event. When the token is used to enter, the turnstile or bouncer validates it and issues a token that can be used to exit. Conversely, when exiting, the customer receives an entry token that allows them to reenter the venue during the duration of the event. The exit token is valid for a longer period than the entry token, allowing all attendees to exit at the end of the event, while preventing people from reentering. This solution ensures that only attendees with valid tickets can enter and prevents ticket sharing fraud, while maintaining the privacy of attendees.

Event deposit system

Deposit systems at events, such as festivals, typically hand out special tags when purchasing an item with a deposit. These tags need to be returned along with the deposit item in order to redeem the deposit. However, the tags commonly used in these systems are prone to duplication or unauthorized acquisition, thereby increasing the risk of fraud. These tags could be substituted with tokens.

Unlinkable gifts

Tokens can facilitate the gifting of digital goods while maintaining both the privacy of the giver and the recipient. To illustrate, a newspaper subscription might permit the sharing of a specified number of articles each week through gift tokens. The tokens permit customers to gift articles anonymously, thereby preventing the merchant from having any information on how the articles where shared.

1.5. Limitations

While the proposed solution offers potential advantages for customers and merchants in terms of privacy and usability, it is important to be aware of its inherent limitations.

1. **Browser fingerprinting.** Despite the privacy enhancements introduced by this solution, it is still possible for merchants to track users through their browser characteristics. To circumvent such tracking, users are recommended to use solutions such as Tor [5].

2. **Business models that rely on profiling customers.** Merchants employing business models that are heavily reliant on profiling customers for targeted advertising or personalized services are unlikely to be motivated to switch to the proposed solution. As a preliminary step, these merchants could consider offering both kinds of subscriptions, with the privacy-preserving option at a price increase. Such a step would clearly communicate the monetary value of profiling consumption habits, making these business models more transparent.

3. **Free trials.** Given that users are anonymous and their actions unlinkable, limiting customers from accessing a free trial more than once is not possible. To circumvent this limitation, merchants may offer paid trials with a duration that is a small fraction of that of regular subscriptions. Furthermore, the solution can be used to model upselling strategies. For instance, purchasing a single magazine article issues an up-sell token to the customer. Should the customer subsequently wish to purchase a subscription, they may utilize the up-sell tokens to credit the expenditure on the articles to the subscription price.
4. **Forced termination of single subscription.** Since user are anonymous, the subscription of a particular user cannot be terminated by the merchant. If such a property is desired, solutions such as [6] are available.

1.6. Notation and terminology

Throughout this thesis, TypeScript[7] is used to define interfaces and data structures. Common data types used across these definitions are listed in Subsection 3.6. All symbols used in the definition of protocols are defined in List of Symbols on page ix.

[7] `https://www.typescriptlang.org/`

2. Background

This chapter offers an overview of the background that serves as the foundation for the work presented in this thesis. In addition, it lists and categorizes existing solutions and previous work done in the field of anonymous subscriptions and discounts.

2.1. Anonymity and unlinkabilty

In the realm of cryptographic systems, anonymity and unlinkability are two fundamental concepts that often surface in discussions around privacy and security. Both terms play a crucial role in ensuring user privacy, but it's important to know how they differ and that they cannot be used interchangeably.

Anonymity

For this thesis, the widely accepted definition of anonymity given by Pfitzmann and Hansen [7] is used.

Definition 1. *Anonymity of a subject means that the subject is not identifiable within a set of subjects, the* anonymity set*.*

As stated by Fusenig and Staab in [8], "Anonymity is usually related to an action α". In other words, anonymity ensures that an observer of a system cannot identify an actor that has performed α within the *anonymity set*. The anonymity set consists of all actors that could have possibly performed α. Therefore we can conclude that the greater the anonymity set size, the stronger the anonymity property. Additionally it's worth noting that anonymity is not considered a binary property. An attacker can have information that leads them to conclude that certain subjects within the anonymity set are more likely to be the actor of α. This probability is described by the concept of the *degree of anonymity*, defined in a information theoretic approach in [9] and [10] respectively.

Unlinkability

Related to anonymity is the property of unlinkability. Again Pfitzmann and Hansen [7] provide the definition of unlinkability that is used throughout this thesis.

Definition 2. *Unlinkability of two or more items of interest (IOI) from an attackers perspective means that within the system, the attacker cannot sufficiently distinguish whether these IOIs are related or not.*

Instead of using the notion of *items of interest*, this thesis refers to *actions* when talking about the property of unlinkability. Therefore, two actions α_0 and α_1 in a system are unlinkable, if an observer cannot find any relation between them. The set of all actions is called the *unlinkability set*.

Following Definition 2, it can be concluded that actions in a system where actors use fixed pseudonyms are always linkable, which in turn leads to a significant reduction in the degree of anonymity. Therefore, solutions relying on static public keys as pseudonyms, such as certain blockchain-based solutions, are inherently linkable.

2.2. Blind signatures

Blind signatures are the foundation of all Chaumian electronic cash solutions as originally proposed in [3]. As Chaum pointed out, blind signatures can be illustrated by the analogy of a carbon paper lined envelope. If you put a piece of paper inside the envelope, seal it and sign from the outside, the signature will be transferred to the paper on the inside. Blind signatures are a powerful cryptographic primitive that enabled a whole new category of secure and private protocols.

The GNU Taler project is cipher agile, allowing administrators to select between two blind signing schemes. The first option is RSA, as originally described in by Chaum in [9], while the second is Clause Blind Schnorr, as described in [11].

2.3. GNU Taler

GNU Taler (Taxable Anonymous Libre Electronic Resources) is an electronic payment system based on open standards, free software and modern cryptography. While it provides anonymity for the buyers, sales incomes of merchants are transparent and therefore taxable. In contrast to proof-of-work based blockchain solutions, it is efficient in computation, resulting in low energy consumption.

The GNU Taler payment system consists of three core components: *exchange*, *merchant* and *wallet*. The exchange allows customers to withdraw digital coins. These coins are then stored in the wallet of the customer. The wallet can be a native iOS/Android mobile app or a web-based application such as a browser extension. To buy something, the wallet will send a signed payment to a merchant. Merchants will validate this payment and check with the exchange to make sure the received coins are valid. Note that only the merchant and the wallet are relevant in the context of implementing tokens.

The merchant component can be operated by a physical store or an online shop.

The merchant is split into the backend and back office. The core component is the merchant backend, which is a server providing a RESTful API and interacting with a Postgres database. The RESTful API is consumed by the merchant back office, wallets, e-commerce solutions, and optionally a Point of Sale (POS) system. Depending on the use case, the merchant back office, e-commerce solution or POS system is responsible to create orders that can be payed by wallets. Endpoints of the merchant backend that are accessible to the wallet are referred to as *public*. Public endpoints are exposed to the internet, since they have to be accessible by any wallet, while *private* endpoints are only accessible by authorized clients such as the merchant back office, e-commerce solution or POS terminal.

2.4. Existing solutions

Existing solutions for the problem of privacy-preserving subscriptions can be categorized into the following groups.

- **Digital cash and k-times anonymous authentication.** Solutions that rely on digital cash as described by Chaum in [3] or k-times anonymous authentication as described in [12] are no efficient solution to the problem stated in this thesis, since the storage requirement for the user are proportional to the subscription length. Furthermore, subscriptions are easily sharable among large groups of users.
- **Unlikable tokens.** To overcome the problems mentioned in the previous group, Stubblebine, Syverson, and Glodschlag porposed a solution that builds upon the concept of blindly signed single-use tokens in [13]. That work is the closest to the one done in this thesis. The key management process and the expiration of subscriptions are not clearly defined in the protocol. Furthermore, the protocol is not implemented as a practical solution.
- **Group signatures.** Unlinkable tokens can be modeled using group signature schemes. They allow a member of a group to sign messages anonymously in the name of the group. According to Ramzan and Ruhl [14]:

  > The main problem with the application of group signatures in general (and also in [15]) is the fact that the group manager (or a designated revocation authority) is able to revoke the anonymity of a client. Moreover, since the schemes are randomized and unlinkable, there is no effective way of limiting the length of a subscription without recovering the identity of users.

- **Verifiable credentials and randomized tokens.** The solution proposed by Blanton in [6] uses the Camenisch-Lysyanskaya (CL) signature scheme to employ randomized tokens for subscriptions. It allows to more granularly define the expiration date of tokens without lowering the degree of anonymity. This benefits comes at the cost of performance, since the zero-knowledge proofs of knowledge employed in the protocol are much more expensive to compute compared to the solution proposed in this thesis. Exploring the feasibility of a verifiable credential scheme for the solution proposed in this thesis is a topic for future work, as further explained in Section 5.4.

In addition, there is a considerable body of previous work on the problem of privacy-preserving discounts and coupons. This includes, for example, the work presented in [16], [17], and [18]. These proposals vary in the degree of anonymity they afford, with some offering a certain degree of resistance to abuse.

The advantage of the coupon solution presented in this thesis is that it is fully unlinkable and can be integrated into a larger solution that can cover a variety of other use cases, such as subscriptions. Consequently, merchants are able to utilize a single solution to facilitate both subscription and discount management.

3. Design

The protocol is comprised of two primary parties: the *merchant* and the *customer*. The merchant offers a subscription to the customer who can then use the subscription to access the merchant's subscription goods, such as articles or movies.

The merchant's primary concern is the unforgeability of subscriptions, which ensures that customers cannot forge tokens to obtain unauthorized access to goods. Furthermore, the merchant desires a solution that is cost-effective to operate, encompassing both transactional and operational expenses associated with the technical infrastructure. Additionally, the merchant aims to restrict the sharing of subscriptions among large groups.

In contrast, the customer desires a user-friendly solution and is motivated to minimize the disclosure of personal information, limiting the collection of sensitive data. Moreover, they want to avoid auto-renewing subscriptions that are difficult to cancel [4].

Upon purchasing a subscription, the merchant issues an anonymous, single-use token to the customer. This is illustrated in Figure 3.1. To access a subscription good, the customer transfers this token to the merchant for verification. If the token is deemed valid, the merchant grants the customer access to the requested good and issues a new token, thereby initiating a chain of tokens where only the most recently issued token is valid. The unlinkability of tokens is ensured by employing a blind signing protocol for token issuance.

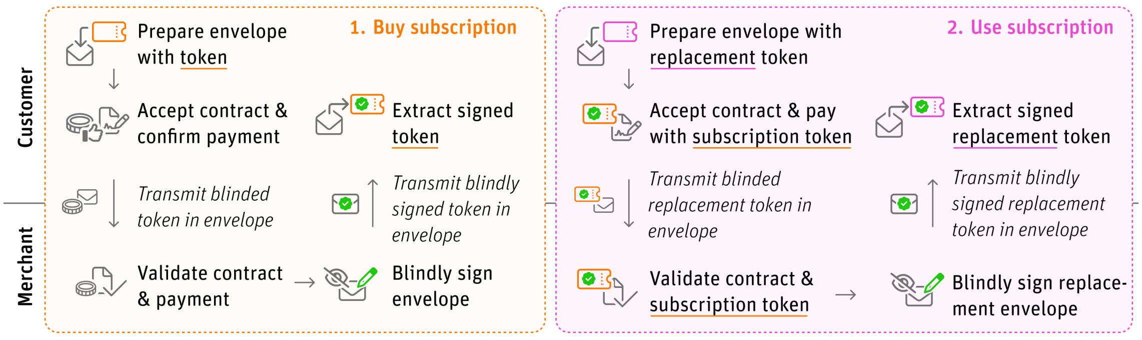

Customer
Merchant
1. Buy subscription
Prepare envelope with token
Accept contract & confirm payment
Extract signed token
Transmit blinded token in envelope
Transmit blindly signed token in envelope
Validate contract & payment
Blindly sign envelope
2. Use subscription
Prepare envelope with replacement token
Accept contract & pay with subscription token
Extract signed replacement token
Transmit blinded replacement token in envelope
Transmit blindly signed replacement token in envelope
Validate contract & subscription token
Blindly sign replacement envelope

Figure 3.1: Schematic flow of buying and using a token-based subscription.

To configure tokens, the merchant sets up a *token family* per subscription. The token family contains metadata such as the relative duration (e.g., 30 days or 1 year) and the name of a subscription. Once a subscription is purchased for the first time, a signing key (henceforth referred to as the *token issue key* or $(k_{\text{issue}}, K_{\text{issue}})$) is generated for the requested token family and subscription period. This key is assigned a validity start and end date, meaning tokens signed by this key will only be valid during that timeframe. A token family can have many signing keys. This relationship is illustrated in Figure 3.2.

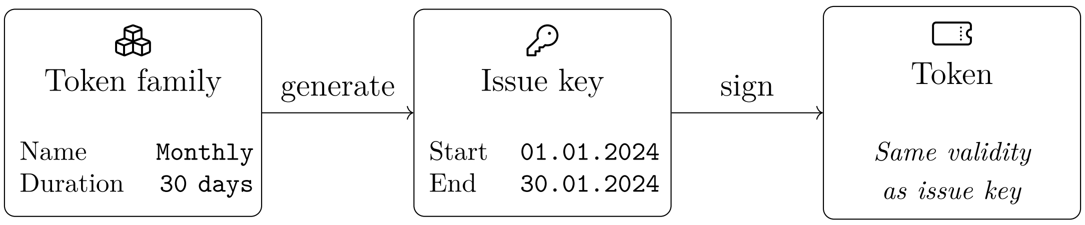


Figure 3.2: Relationship between token families, token issue keys and tokens.

3.1. Token families and start date rounding

Merchants may configure token families to enable their customers to utilize tokens. A token family may represent either a subscription or a discount, as apparent by the `kind` field in the code snippet in Figure 3.4. It is accompanied by a human-readable name and description, which will be displayed to the customer in their wallet. The slug of a token family is a URL-safe identifier that is unique across all token families configured in the given merchant instance.

```
interface TokenFamily {
  slug: string;
  name: string;
  description: string;
  description_i18n?: { [lang_tag: string]: string };
  valid_after?: Timestamp;
  valid_before: Timestamp;
  duration: RelativeTime;
  rounding: RelativeTime;
  kind: TokenFamilyKind;
}
```

Figure 3.3: Token family specification.

```
enum TokenFamilyKind {
  Discount = "discount",
  Subscription = "subscription",
}
```

Figure 3.4: Token families can represent discounts or subscriptions.

In addition to other metadata, a token family also specifies the validity (`duration` field) and rounding (`rounding` field) durations. The validity duration specifies the period of time that a token should remain valid after issuance. The rounding duration is employed to round down the start date of the token. As an example: a merchant configures a monthly subscription with a duration of 30 days and a rounding duration of a day. This implies that the start date of any newly purchased subscription will be rounded down to the start of the day. Meaning a subscription purchased on June 1 at 15:32 will have a start date of June 1 at 00:00.

The rounding process is a crucial aspect of maintaining the anonymity of token holders. Token holders are only anonymous within the group of customers that own a token with the same validity period, the anonymity set. When determining the rounding duration for use cases, it is essential to carefully evaluate the duration to ensure adequate anonymity for all token holders. Consider a service with only ten subscribers. If the service offers a subscription with a rounding duration of one day, it is highly probable that each customer will purchase their subscription on a different day. In such a scenario, the tokens could potentially be used to identify individual users and link their actions, thereby compromising their privacy (for further details, please refer to Section 5.2).

In addition to its role in ensuring privacy, the rounding duration is also useful for modeling subscriptions that are always valid for the duration of the current calendar year, regardless of date of purchase. For such use cases, the merchant configures a token family with a validity duration of one year and a rounding duration of one year.

For more complex subscriptions, such as the Swiss motorway vignette[1], an additional offset property would have to be added to token families. This is because the motorway vignette is valid for 14 months starting on December 1. Subsequently, the merchant could configure a token family with a duration of 14 months, a rounding duration of one year and an offset of one month, thereby establishing the desired validity period for issued tokens.

3.2. Contract terms

When a customer desires to purchase an item from a merchant, the merchant presents their wallet with an *order*. The order contains information regarding the total price, the products included, transaction fees, delivery dates, and other metadata. If the customer agrees with the terms outlined in the order, the wallet signs the order, thereby transforming it into a *contract*.

Since the changes made to the contract terms data structure are not backward compatible, this new contract terms format is referred to as *version 1*. Throughout this thesis, orders or contract terms that use or issue tokens will be referenced as *v1*

[1]`https://www.ch.ch/en/vehicles-and-traffic/how-to-behave-in-road-traffic/motorway-vignette/`

contract terms or orders.

To facilitate the use of tokens, the existing contract terms of GNU Taler (henceforth referred to as *v0*) are adapted to allow for multiple ways how a contract can be paid. This is accomplished by a new array of contract choices, as shown in the code snippet in Figure 3.8. Prior to signing an order, the customer is presented with a selection of these choices. Each choice comprises of a gross price, maximum deposit fees accepted by the merchant, inputs, and outputs. Inputs represent the tokens that the wallet must provide in order to fulfill the contract, while outputs specify which tokens will be issued upon successful payment of the contract.

The variability in price associated with different choices enables a wide range of new possibilities. For instance, an article can either be purchased at a fixed price or accessed for free through the use of a subscription token. The process of using a subscription can be represented by a choice with a gross price of zero, a subscription token as an input and a subscription token of the same token family as an output. Upon encountering a contract with such a choice, a wallet is able to pre-select this choice, potentially even making an automatic selection, given that it is essentially free. This approach facilitates a seamless user experience, comparable to that of conventional subscription services. In addition, the price of each choice can have a different currency, allowing the creation of multi-currency contracts.

To efficiently store metadata and keys of relevant token families, the v1 contract terms introduce a `token_families` map within the contract. The token family slug is used as a key for accessing information within that map, thereby ensuring constant time access.

```
interface ContractTermsV1 {
  // other fields omitted for brevity
  version: 1;
  choices: ContractChoice[];
  token_families: { [token_family_slug: string]: ContractTokenFamily };
}

interface ContractChoice {
  amount: Amount;
  inputs: ContractInput[];
  outputs: ContractOutput[];
  max_fee: Amount;
}
```

Figure 3.5: Choices in contract terms.

As shown in Figure 3.6, input and output tokens each consist of the token family slug, the validity start date, and a number indicating how many tokens should be issued or are required. The validity start date is used to identify the corresponding signing key of the token family.

```
interface ContractInput {
  type: "token";
  token_family_slug: string;
  valid_after: Timestamp;
  number: Integer;
}

interface ContractOutput {
  type: "token";
  token_family_slug: string;
  valid_after: Timestamp;
  number: Integer;
}
```

Figure 3.6: Contract choice inputs and outputs.

It should be noted that inputs and outputs were designed in a generic manner with the intention of enabling other types in the future. For instance, a donation could generate a privacy-preserving tax receipt, or a contract could yield coins in a different currency, thereby enabling currency conversion.

All token family information relevant to the wallet is included in the contract terms, as shown in Figure 3.7. Additionally, a boolean field is included to indicate whether the token family is critical or not. The wallet has to understand all critical token families of a contract to process it. The merchant backend requires token envelopes for all critical output tokens, while envelopes are optional for non-critical tokens. Subscription tokens are considered critical, while discounts are not.

The structure of the `details` field of a token family is dependent on whether to belongs to a subscription or discount token family.

For subscription token families, the `details` field contains a list of URLs labelled `trusted_domains`. This list indicates which URLs a token can be safely used on. This implies that the issuer warrants that these sites will re-issue subscription tokens if the order so states. It also implies that wallets can trust these sites to employ the auto-usage of subscription tokens. Without `trusted_domains`, attackers could trick wallets into employing auto-usage to steal their subscriptions or obtain information on the subscriptions held by the wallets.

Similarly, discount token families have a list of URLs labelled `expected_domains`. In this context, the list indicates on which URLs the given token family is *intended* to be used. However, other sites *may* still request wallets to use tokens of this family, although wallets should warn users in such cases. This flexibility allows merchants to attach different semantics to discount tokens of a competitor, such as granting a 30% discount using the competitor’s 20% discount token.

Both of these URL lists may contain an asterisk as a wildcard selector for any URL.

```
interface ContractTokenFamily {
  name: string;
  description: string;
  description_i18n?: TranslatedString;
  keys: TokenIssueKey[];
  details: SubscriptionTokenFamilyDetails | DiscountTokenFamilyDetails;
  critical: boolean;
}

interface SubscriptionTokenFamilyDetails {
  kind: "subscription";
  trusted_domains: string[];
}

interface DiscountTokenFamilyDetails {
  kind: "discount";
  expected_domains: string[];
}
```

Figure 3.7: Token families in contract terms.

Figure 3.8 shows that token issue keys have a validity period and can be of two different ciphers: RSA [9] or Clause Schnorr [11]. The `valid_after` of an input or output is identical to that of the corresponding issue key, thus enabling a wallet to match them.

```
type TokenIssueKey = TokenIssueRsaKey | TokenIssueCsKey;

interface TokenIssueRsaKey {
  cipher: "RSA";
  rsa_pub: RSAPublicKey;
  valid_after: Timestamp;
  valid_before: Timestamp;
}

interface TokenIssueCsKey {
  cipher: "CS";
  cs_pub: CsPublicKey;
  valid_after: Timestamp;
  valid_before: Timestamp;
}
```

Figure 3.8: Token issue keys per token family in contract terms.

3.3. Keys and their lifecycle

The proposed solution comprises two distinct types of keys. One is managed by the wallet, while the other is managed by the merchant.

Token use key is an EdDSA key pair generated and stored in the wallet of a customer. While the private key is used to create signatures and remains private throughout its lifespan, the public key is revealed once a token is used in a transaction.

Token issue key is an RSA or Clause Schnorr key pair managed by the merchant. It is used to issue and validate tokens. The validity timeframe of a token is coupled to the validity timeframe of the token issue key it was signed with.

Key generation

Figures 3.9, 3.10 and 3.12 provide a graphical overview of the generation of keys, while Figure 3.11 shows of the merchant backend uses an existing key.

1. Figures 3.9: The merchant back office sends a request to the merchant backend RESTful API to create a new token family. Note that this request does not generate a token issue key yet.
2. Figure 3.10: The back office sends another request to the backend to prepare an order that issues a token of the newly configured token family. The backend validates the referenced token family and checks its database for the existence of a token issue key. Since there is no key in the merchants database yet, it generates a new private key, derives the public key and calculates its hash. By default, this is an RSA key but Schnorr signatures can also be configured. The validity timeframe of the key is based on the rounding and validity duration of the token family. Finally it stores the generated key with the corresponding metadata in the database.
3. Figure 3.12: Once the order is prepared, the customer can instruct their wallet to pay for the order. The wallet will generate a new EdDSA token use private key and derive its public key. It will send the blinded EdDSA public key along with the required amount of coins to the merchant backend. The backend will validate the payment and, on success, sign the blinded EdDSA token use public key with the merchant token private key. Once the backend responds with the blind signature, the wallet will store this along the corresponding token use key.

In order to streamline the key management lifecycle, token issue keys are generated only once required. This approach offers several advantages in the context of merchant implementation. By generating keys synchronously and on demand, it avoids unnecessary overhead associated with pre-generating keys, thus reducing the time and resources required for key management.

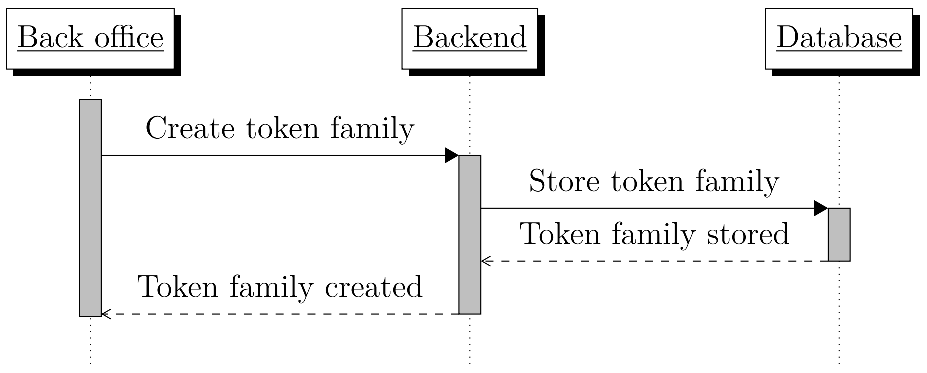


Figure 3.9: Token family creation.

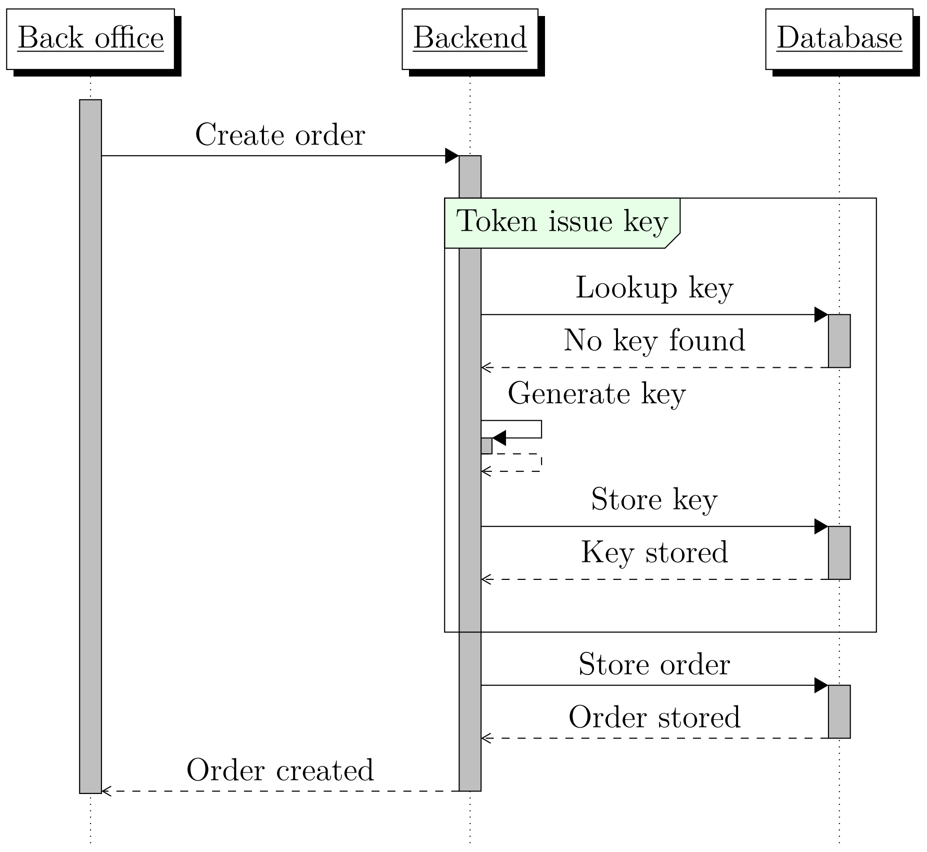


Figure 3.10: Initial token issue key generation triggered by the creation of an order.

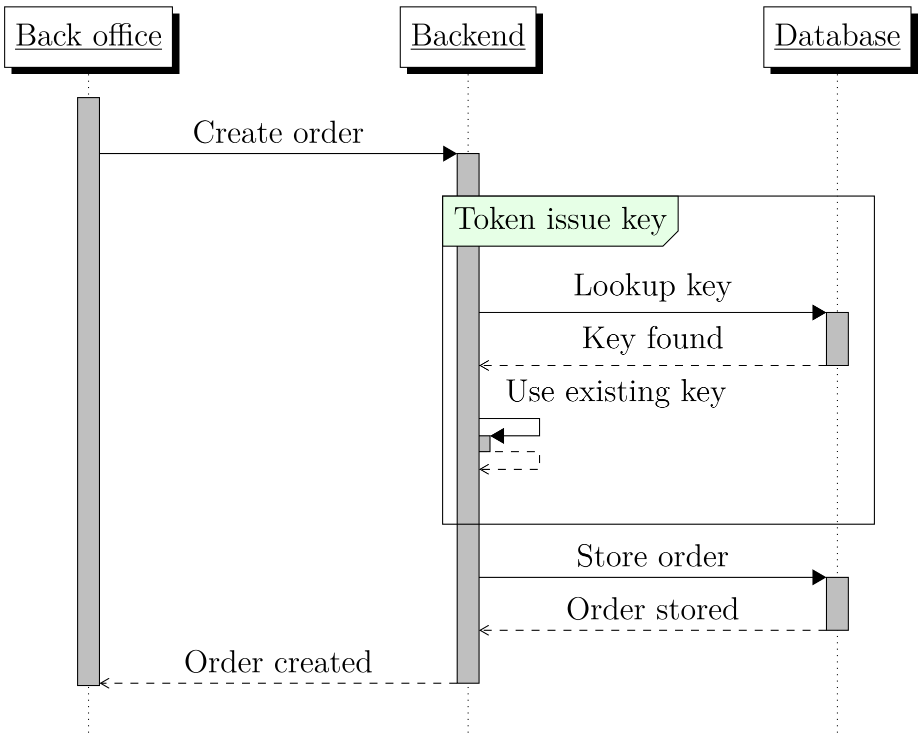


Figure 3.11: Token issue key process when the key already exists in the database.

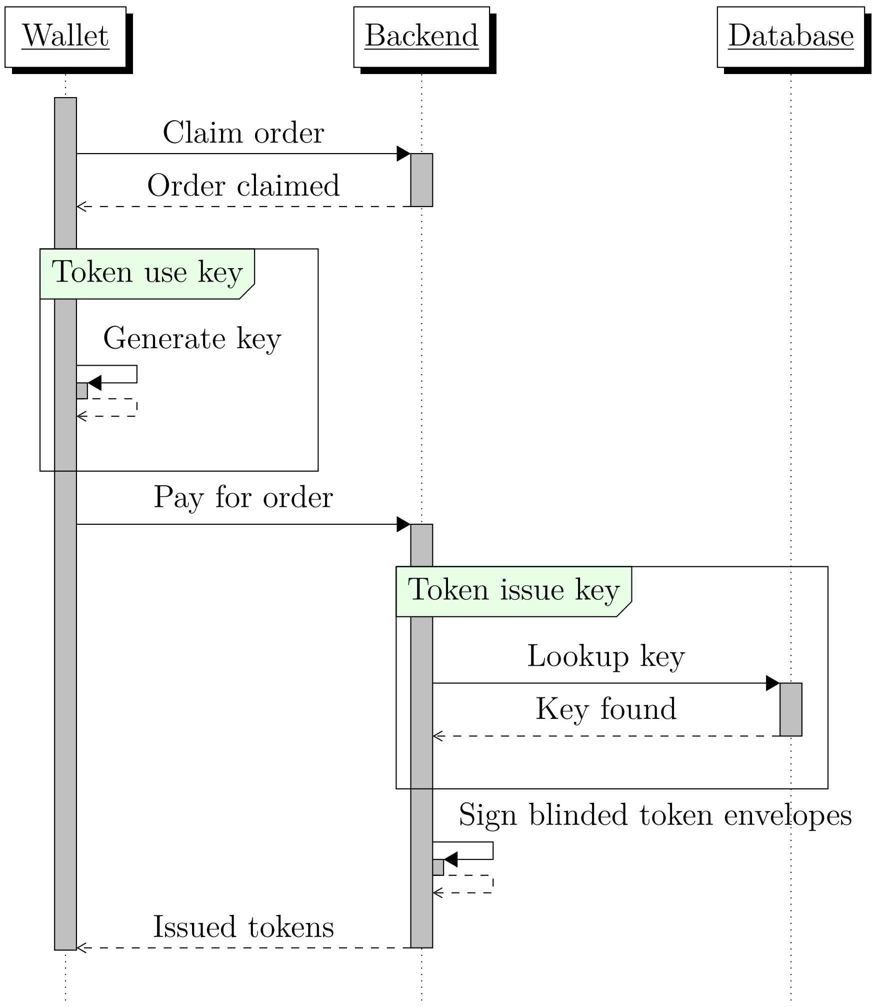


Figure 3.12: Payment process for an order using input and output tokens.

3.4. Cryptography

The proposed solution employs two distinct types of keys (see Section 3.3), resulting in two distinct types of signatures. The first type of signature is a blind signature generated by the merchant backend. This signature serves to verify that a token was issued by the merchant and is henceforth refered to as *token issue signature*. The second type of signature is generated by the wallet and serves to guarantee that a wallet is authorized to use a specific token for a particular contract. These signatures are referred to as *token use signature*.

To ensure consistency across different systems of the network, it is imperative that all cryptographic functions be executed on binary data in network byte order (big-endian). Unless explicitly specified otherwise, hashes are to be performed using the SHA-512 hash function, as defined in [19], which results in a 512-bit hash. Figure 3.13 represents the data structure utilized to represent hash values in C code. Prior to hashing JSON arrays or objects, clients must normalize them by sorting the keys and compacting the resulting output.

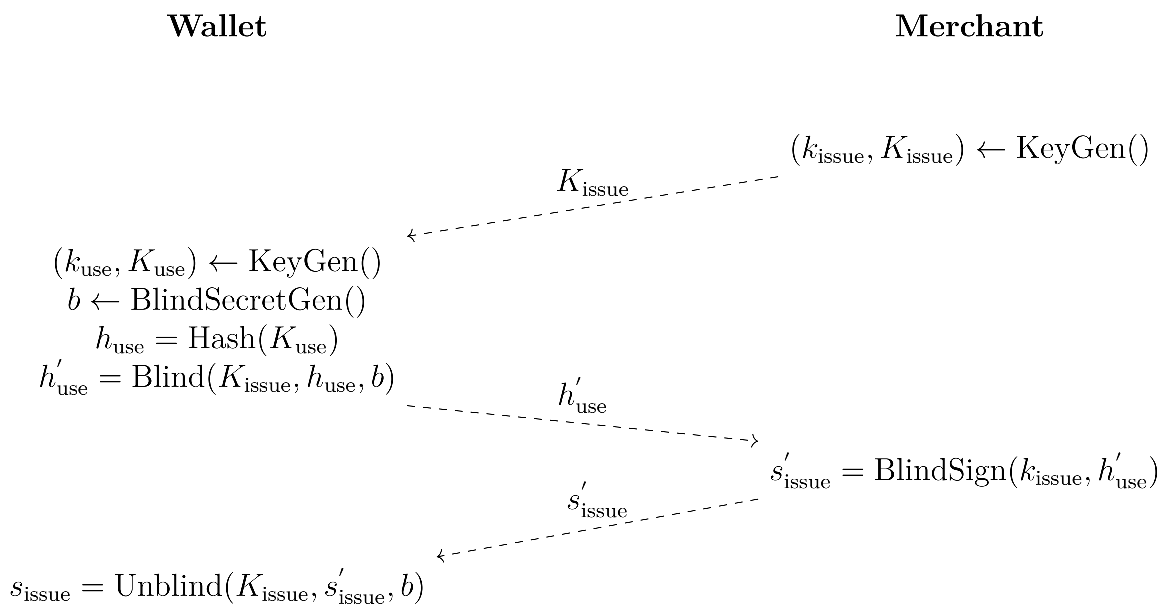
```
struct GNUNET_HashCode
{
  uint32_t bits[512 / 8 / sizeof(uint32_t)];  /* = 16 */
};
```

Figure 3.13: Wrapper struct used to represent a 512-bit hash value.

Token issuance protocol

The merchant backend can be configured to use either blind RSA [20] or Clause Blind Schnorr [11] signature schemes, while RSA signatures with 2048-bit key length is the default.

To issue new tokens, the merchant backend blindly signs the hash of a token use public key provided by a wallet and then transmits the resulting signature back to the wallet (see Figure 3.14). Since the wallet knows the blinding secret, it can then unblind the signature, which will result in a token issue signature. Due to the nature of blind signatures, the issue signature is unknown to the merchant until it is revealed for the first time in the process of using a token. Upon seeing a token issue signature for the first time, the merchant backend will store it in the database to prevent double spending. Once a token issue key has expired, the merchant can forget the used tokens associated with that key. This process ensures that the database does not grow indefinitely.



Wallet
Merchant
$(k_{\text{issue}}, K_{\text{issue}}) \leftarrow \text{KeyGen}()$
K_{issue}
$(k_{\text{use}}, K_{\text{use}}) \leftarrow \text{KeyGen}()$
$b \leftarrow \text{BlindSecretGen}()$
$h_{\text{use}} = \text{Hash}(K_{\text{use}})$
$h'_{\text{use}} = \text{Blind}(K_{\text{issue}}, h_{\text{use}}, b)$
h'_{use}
$s'_{\text{issue}} = \text{BlindSign}(k_{\text{issue}}, h'_{\text{use}})$
s'_{issue}
$s_{\text{issue}} = \text{Unblind}(K_{\text{issue}}, s'_{\text{issue}}, b)$

Figure 3.14: Token issuance protocol.

Token use protocol

All signatures created by the wallet use the generalized version [21] of the EdDSA signature scheme [22].

Figure 3.16 shows a graphical representation of the token use protocol. To use a token in a transaction, the wallet must first sign a *token use request*. The request is signed with the *token use private key*, resulting in a token use signature. Figure 3.15 shows the data structure of this request. As is standard for all EdDSA signable data structures within the GNU Taler ecosystem, the token use request starts with a header. This header contains four bytes representing a unique signature purpose number, as defined in Table 3.1. This number is used to enforce distinct signatures for disparate purposes, despite identical signed data. The second piece of data in the header is the size (in bytes) of the to be signed data to prevent length extension attacks. Following the header are two hash values, each 16 bytes in length. The first is the hash of the contract terms JSON object agreed upon with the merchant backend. The second is an object containing additional data provisioned by the wallet, known as the *wallet data*. This wallet data object contains the index of the selected contract choice and the token envelopes provided for this transaction, thereby committing the wallet to the specified envelopes.

The resulting token use signature, in conjunction with the token issue signature acquired during issuance, are transmitted to the merchant backend. The backend then validates both signatures and accepts the token if they are deemed valid.

```
struct TALER_TokenUseRequestPS
{
  uint32_t purpose;
  uint32_t size;
  struct GNUNET_HashCode h_contract_terms;
  struct GNUNET_HashCode h_wallet_data;
};
```

Figure 3.15: Data structure for token use request.

Purpose	Description	Value
`TALER_SIGNATURE_WALLET_TOKEN_USE`	Token use private key confirms the usage of a token on a pay request.	1222

Table 3.1: Constants used in purpose field of header for EdDSA signatures.

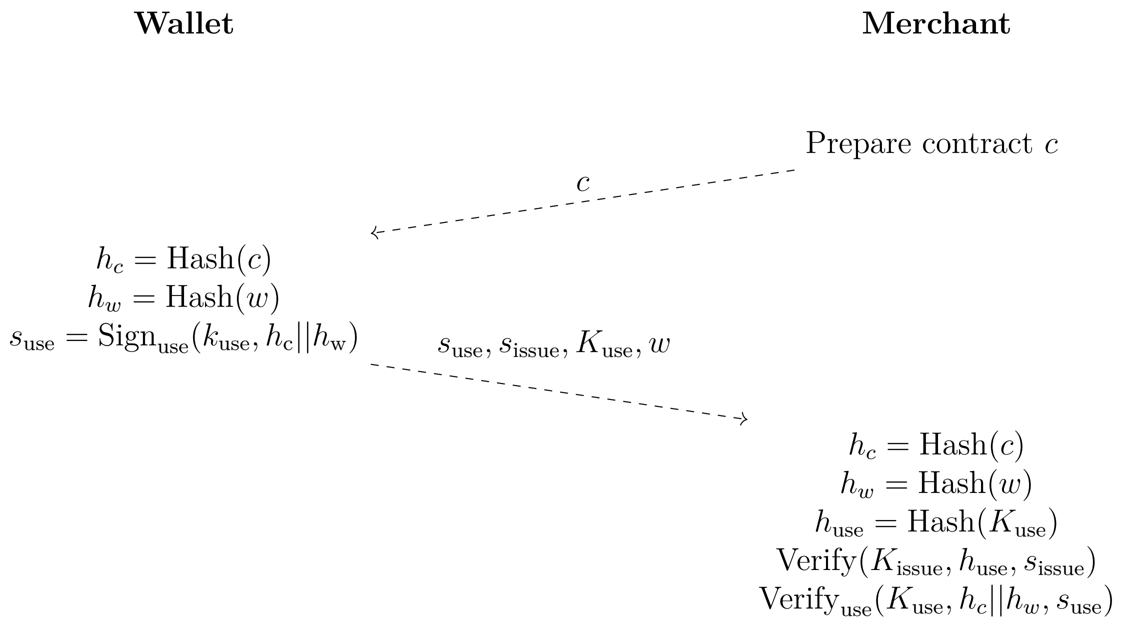

Wallet
Merchant
Prepare contract c
c
$h_c = \text{Hash}(c)$
$h_w = \text{Hash}(w)$
$s_{\text{use}} = \text{Sign}_{\text{use}}(k_{\text{use}}, h_{\text{c}}||h_{\text{w}})$
$s_{\text{use}}, s_{\text{issue}}, K_{\text{use}}, w$
$h_c = \text{Hash}(c)$
$h_w = \text{Hash}(w)$
$h_{\text{use}} = \text{Hash}(K_{\text{use}})$
$\text{Verify}(K_{\text{issue}}, h_{\text{use}}, s_{\text{issue}})$
$\text{Verify}_{\text{use}}(K_{\text{use}}, h_c||h_w, s_{\text{use}})$

Figure 3.16: Token use protocol with s_{issue} and K_{use} as defined in Figure 3.14.

Combined token use and issuance protocol

The token use and issuance protocols can be combined to permit the use of a token in the same transaction as a new token is issued. This combined protocol serves as the foundation for use cases with constantly rotating tokens such as subscriptions. Figure 3.17 illustrates this combined protocol, where the wallet is issued a token n which it then uses on a transaction that issues a new token $n + 1$.

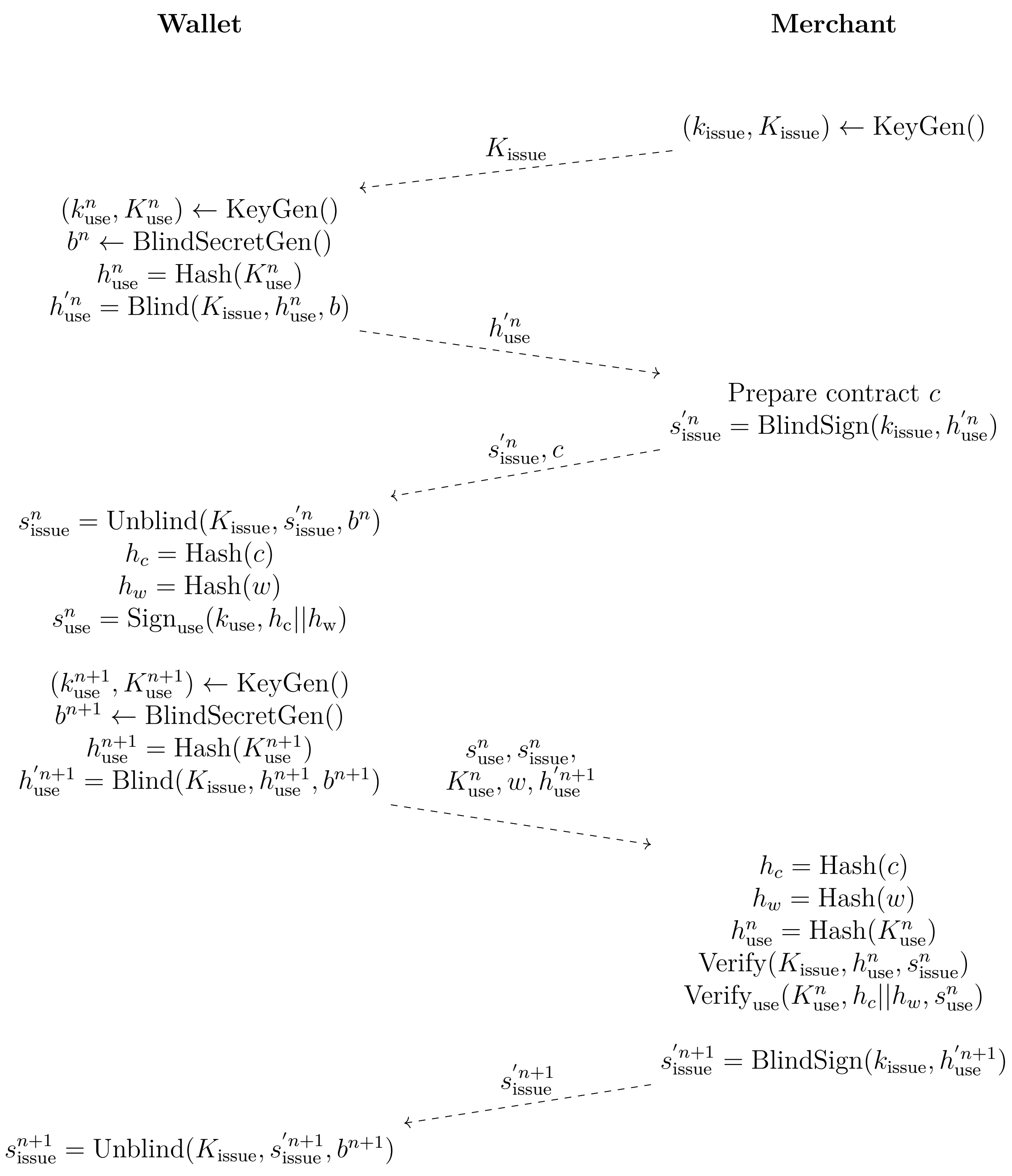

Wallet
Merchant
$(k_{\text{issue}}, K_{\text{issue}}) \leftarrow \text{KeyGen}()$
K_{issue}
$(k^n_{\text{use}}, K^n_{\text{use}}) \leftarrow \text{KeyGen}()$
$b^n \leftarrow \text{BlindSecretGen}()$
$h^n_{\text{use}} = \text{Hash}(K^n_{\text{use}})$
$h'^n_{\text{use}} = \text{Blind}(K_{\text{issue}}, h^n_{\text{use}}, b)$
h'^n_{use}
Prepare contract c
$s'^n_{\text{issue}} = \text{BlindSign}(k_{\text{issue}}, h'^n_{\text{use}})$
s'^n_{issue}, c
$s^n_{\text{issue}} = \text{Unblind}(K_{\text{issue}}, s'^n_{\text{issue}}, b^n)$
$h_c = \text{Hash}(c)$
$h_w = \text{Hash}(w)$
$s^n_{\text{use}} = \text{Sign}_{\text{use}}(k_{\text{use}}, h_{\text{c}}||h_{\text{w}})$
$(k^{n+1}_{\text{use}}, K^{n+1}_{\text{use}}) \leftarrow \text{KeyGen}()$
$b^{n+1} \leftarrow \text{BlindSecretGen}()$
$h^{n+1}_{\text{use}} = \text{Hash}(K^{n+1}_{\text{use}})$
$h'^{n+1}_{\text{use}} = \text{Blind}(K_{\text{issue}}, h^{n+1}_{\text{use}}, b^{n+1})$
$s^n_{\text{use}}, s^n_{\text{issue}}, K^n_{\text{use}}, w, h'^{n+1}_{\text{use}}$
$h_c = \text{Hash}(c)$
$h_w = \text{Hash}(w)$
$h^n_{\text{use}} = \text{Hash}(K^n_{\text{use}})$
$\text{Verify}(K_{\text{issue}}, h^n_{\text{use}}, s^n_{\text{issue}})$
$\text{Verify}_{\text{use}}(K^n_{\text{use}}, h_c||h_w, s^n_{\text{use}})$
$s'^{n+1}_{\text{issue}} = \text{BlindSign}(k_{\text{issue}}, h'^{n+1}_{\text{use}})$
s'^{n+1}_{issue}
$s^{n+1}_{\text{issue}} = \text{Unblind}(K_{\text{issue}}, s'^{n+1}_{\text{issue}}, b^{n+1})$

Figure 3.17: Combined token use and issuance protocol.

3.5. Architecture

The GNU Taler payment system is comprised of three core components: exchange, merchant, and wallet. The merchant component consists of two distinct services: the backend and the back office.

The merchant backend is responsible for communicating with the exchange and exposes a RESTful API that is consumed by wallets and the merchant back office.

The merchant back office is a user interface implemented as a Single Page Application (SPA) JavaScript application running in the browser. It allows a merchant to manage their connected bank accounts, product inventory, orders and token families. In production deployments, the merchant would typically also have an online shop or a POS system that interacts with the merchant backend to create new orders and present them to wallets for payment. For that purpose the GNU Taler project has plugins that integrate popular e-commerce solutions, such as WooCommerce[2] and Joomla![3].

To implement token-based subscriptions and discounts, the relevant components are the wallet, merchant back office and the merchant backend including its database. The RESTful API of the merchant backend was extended to support tokens. Additionally, the merchant database had to be extended. To configure token families, the merchant back office was extended with new pages that allow the creation, editing, and deletion of token families.

To allow customers to use tokens, the wallet has to be adapted to use these new APIs, which is subject to future work.

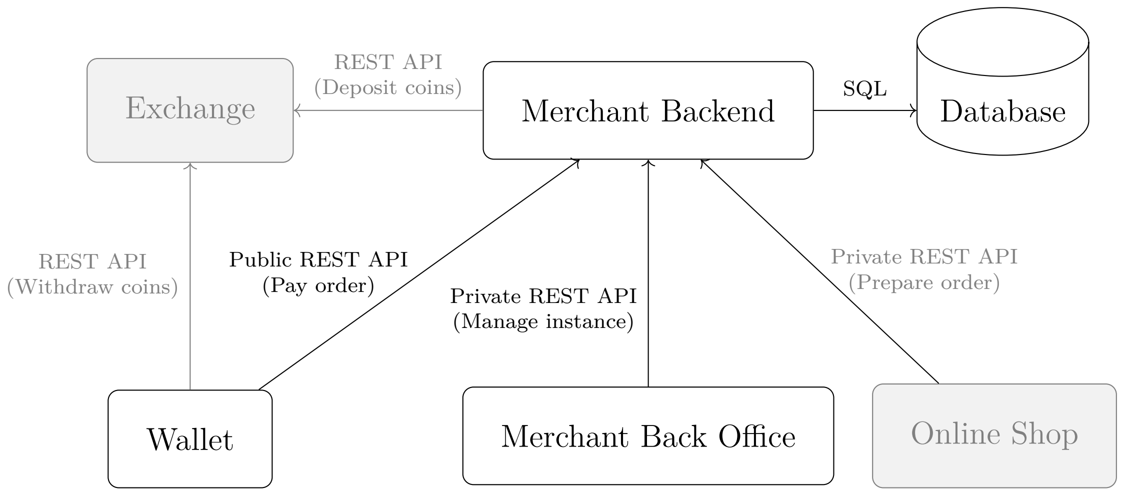


Figure 3.18: GNU Taler architecture overview.

The existing system architecture of GNU Taler (as shown in Figure 3.18) does not have to change to support tokens.

[2]https://git.taler.net/gnu-taler-payment-for-woocommerce.git/

[3]https://git.taler.net/joomla-payage-taler-plugin.git/

3.6. Merchant RESTful API

To enable token support for merchants in the existing GNU Taler codebase, three relevant endpoints have to be implemented in the RESTful API: order creation, order payment, and querying order status. This chapter will focus on the changes made to existing endpoints. For a comprehensive overview of the merchant RESTful API, consult Appendix A.

The merchant backend is capable of supporting multiple merchant instances, thereby enabling a single deployment to serve multiple shops. In order to target a merchant instance other than the default, it is possible to prefix all endpoints listed below with `/instances/INSTANCE_ID`.

Common API data types

The following section provides an overview of the common data types used in both the public and private API. These data types are utilized across all GNU Taler APIs.

Binary data is typically encoded as a Base32 variant, as specified by Cockford[4]. However, one modification is made to enhance the reliability of the Optical Character Recognition (OCR) process: the letter `U` is not excluded from the symbol set, but instead decodes to `V`. For the sake of simplicity, this modified encoding scheme will henceforth be referred to as Base32.

JavaScript does not natively distinguish between integers and other types of numbers. To circumvent this limitation, a special type is employed as an alias to the native number data type. This type indicates that a field is expecting integer numbers. This can be enforced by clients during runtime.

```
type Integer = number;
```

Timestamps are represented by an object with a single key `t_s`. This key can be either an integer, representing the second since the epoch (January 1, 1970), or the special string value `"never"`, which indicates an empty timestamp value.

```
interface Timestamp {
  t_s: number | "never";
}
```

Durations are represented by an object with a single entry, `d_us`, which represents a duration in microseconds. Alternatively, the special string value `"forever"` indicates an infinite duration. The maximum allowed numeric value is $2^{53} - 1$.

```
interface RelativeTime {
  d_us: number | "forever";
}
```

A map is employed to represent strings that can be translated into multiple languages. This map maps a language tag, as specified in [23], to the translated string

[4]https://www.crockford.com/base32.html

value.

```
type TranslatedString {
  [lang_tag: string]: string;
}
```

Currency amounts are serialized as a string in the format `<Currency>:<DecimalAmount>`. To avoid inaccuracies common with floating point numbers, monetary amounts are represented as fixed-precision numbers with eight decimal places. Furthermore, the integer part of the monetary amount may be at most 2^{52} while the fractional part may contain at most eight decimal digits. The currency must be a maximum of 11 characters in length and may only consist of the uppercase and lowercase letters A-Z. Examples of valid amounts include `CHF:10` or `EUR:310`.

```
type Amount = string;
```

Hash values are represented as Base32-encoded strings of the 64-byte binary hash value.

```
type HashCode = string;
```

An EdDSA public key is a point on Curve25519. This point is represented in the standard 256-bit Ed25519 compact format and encoded using Base32. To transmit EdDSA signatures, R and S, as described in [22], of the signatures are Base32 encoded.

```
type EddsaSignature = string;
type EddsaPublicKey = string;
```

Public keys, signatures, and blinded signatures of the RSA signature scheme are transmitted as Base32-encoded strings of their binary representation.

```
type RsaPublicKey = string;
type RsaSignature = string;
type BlindedRsaSignature = string;
```

In the context of the Clause Blind Schnorr signature scheme, the nonce is represented as a Base32-encoded 32-byte value that must be used only once. Similarly, points on Curve25519 are also Base32-encoded 32-byte values. Blinded challenges are represented by scalar multipliers. The multipliers are 32-byte values used for scalar operations on Curve25519 points and encoded as Base32.

```
type CsNonce = string;
type Cs25519Point = string;
type Cs25519Scalar = string;
```

Private API

The private RESTful API of the merchant backend is exclusively utilized by the merchant back office and is not accessible to wallets. The API exposes endpoints for the management of orders and token families. It features HTTP-based authentication that can be configured on a per-instance basis. The following list enumerates all endpoints that require modification or implementation in order to enable token support in the merchant backend.

GET `/private/orders/ORDER_ID` *Inspect an existing order.*

An existing order can have a status of *paid*, *claimed* or *unpaid*. Only the response for paid orders is listed below, since that is the only part of the API that will change. In the response for paid orders, the merchant backend will include a field labeled `choice_index`, which indicates the selected choice of a wallet. This field is optional and will only be set for v1 orders.

Response

```
interface OrderPaidResult {
  // other fields omitted for brevity
  choice_index?: Integer;
}
```

POST `/private/orders` *Create a new order that a customer can pay for.*

The create order request allows for the specification of an optional array of choices, each choice comprising of its gross price, inputs, and outputs. Each input and output specifies the token family it belongs to and an optional `count` integer, which may be used to specify the number of tokens that will be issued or that are required. The `count` field defaults to one and all input with a count field smaller than one are ignored. To specify the creation of a v1 order, the optional `version` field must be set to the value of `1`. If not set, `version` default to `0`.

When receiving a create order request, the backend queries the database for a token issue key of the token family specified by `token_family_slug` with a matching validity start date. The desired start date is calculated depending on whether the token in question is defined as an input or output.

In the case of an input token, the backend calculates the current time and rounds it down according to the rounding duration of the token family (for further details on the rounding process, please refer to Section 3.1). The validity start date of the token issue key must be equal to the rounded value.

In the case of an output token, the backend selects the matching token issue key based on the provided `valid_after` field. If `valid_after` is not set, the current time is used, analogous to input tokens. This field is also subject to the rounding logic defined in Section 3.1.

If no matching token issue key is found in the database, the backend generates a new key pair for the requested timeframe and token family. This key pair is then stored in the database.

Request

```
interface CreateOrderRequest {
  // other fields omitted for brevity
  order: OrderV1;
}

interface OrderV1 {
  // other fields omitted for brevity
  version: 1;
  choices: OrderChoice[];
}

interface OrderChoice {
  amount: Amount;
  inputs?: OrderInput[];
  outputs?: ContractOutput[];
  max_fee?: Amount;
}

interface OrderInput {
  type: "token";
  token_family_slug: string;
  count?: Integer;
}

interface OrderOutputToken {
  type: "token";
  token_family_slug: string;
  valid_after?: Timestamp;
  count?: Integer;
}
```

POST `/private/tokenfamilies` *Create a new token family.*

A token family is identified by a human-readable identifier called the *token family slug*. This identifier is composed of unreserved URL characters, in accordance with RFC 3986 [24]. Additionally, a token family has a name and a description that can be internationalized. The optional `valid_after` field defaults to the current time of the merchant backend and must not be in the past. The value of the `valid_before` timestamp must be strictly greater than the `valid_after` timestamp. The length of the validity timeframe of an issued token is specified by the `duration` field, while the `rounding` field is used to calculate its start date. Finally, the `kind` field is an enum represented as a string indicating whether a token family represents a subscription or a discount.

```
interface TokenFamilyCreateRequest {
  slug: string;
  name: string;
  description: string;
  description_i18n?: TranslatedString;
  valid_after?: Timestamp;
  valid_before: Timestamp;
  duration: RelativeTime;
  rounding: RelativeTime;
  kind: TokenFamilyKind;
}

enum TokenFamilyKind {
  Discount = "discount",
  Subscription = "subscription",
}
```

PATCH **`/private/tokenfamilies/TOKEN_FAMILY_SLUG`** *Update a token family.*

Once a token family has been created, only a subset of its properties can be updated. When a token family is updated, the changes will only affect orders created after the update. Existing orders will continue to be based on the configuration of the token family prior to the update. This also applies to already issued tokens, which will retain their validity as configured during the issuance.

Request

```
interface TokenFamilyUpdateRequest {
  name: string;
  description: string;
  description_i18n: TranslatedString;
  valid_after: Timestamp;
  valid_before: Timestamp;
  duration: RelativeTime;
}
```

GET **`/private/tokenfamilies`** *List all configured token families.*

This endpoint enumerates all configured token families associated with a given merchant instance. The data returned for each token family is streamlined to minimize the size of network requests and the latency of database queries, while providing all information necessary to render an overview table in the user interface.

Response

```
interface TokenFamiliesList {
  token_families: TokenFamilySummary[];
}

interface TokenFamilySummary {
  slug: string;
  name: string;
  valid_after: Timestamp;
  valid_before: Timestamp;
  kind: TokenFamilyKind;
}
```

GET `/private/tokenfamilies/TOKEN_FAMILY_SLUG` *Inspect a token family.*

This endpoint returns detailed information on a token family specified by the URL parameter `TOKEN_FAMILY_SLUG`. This information includes all fields specified during creation and two additional integers: `issued` and `used`. The first indicates the number of tokens of this family that have been issued, while the latter counts the number of tokens that have been used so far. It is evident that the number of issued tokens must always be greater than or equal to the number of used tokens. Otherwise, an unauthorized party would have been able to issue tokens.

Response

```
interface TokenFamilyDetails {
  slug: string;
  name: string;
  description: string;
  description_i18n?: TranslatedString;
  valid_after: Timestamp;
  valid_before: Timestamp;
  duration: RelativeTime;
  rounding: RelativeTime;
  kind: TokenFamilyKind;
  issued: Integer;
  used: Integer;
}
```

DELETE `/private/tokenfamilies/TOKEN_FAMILY_SLUG` *Delete a token family.*

Deleting a token family removes it and all of its keys from the database, rendering all previously issued tokens that belong to that token family unspendable.

Public API

The public RESTful API of the merchant backend is exposed to the internet and typically utilized by wallets. In contrast to the private API, these endpoints don't feature HTTP-based authentication, but use confidential information related to orders to authenticate wallets.

POST `/orders/ORDER_ID/claim` *Claim an order.*

The initial step in the payment process for a wallet is the claiming of an order. To do so, the wallet must provide a nonce, which serves to identify the wallet that claimed the order. It should be noted that an order can only be claimed once, to prevent wallets from paying for the same order twice. Depending on the configuration of the merchant instance, the claim request must also include an authentication token that was generated during the creation of the order.

A successful claim request will return the contract terms of the order. V1 Orders have the optional `version` field set to 1, which would otherwise default to 0. The v1 contract terms are specified in Section 3.2.

Response

```
interface ClaimResponse {
  // other fields omitted for brevity
  contract_terms: ContractTermsV1;
}
```

POST `/orders/ORDER_ID/pay` *Pay for an order.*

When the wallet pays for a v1 order, it must specify the selected choice in the `choice_index` field of the `wallet_data` object of the pay request. Furthermore, the wallet must provide any required input tokens in the `tokens` field. If the selected choice will issue new tokens, it must also include matching token envelopes in the `token_envs` field. Including the token envelopes is optional for tokens belonging to non-critical token families, such as discount tokens.

The ordering of the `tokens` and `token_envs` arrays is critical. These arrays must be ordered in the same way as their definition in the contract terms. For instance, the envelope at the first position corresponds to the token family at the first position of the `outputs` array of the selected choice. For inputs or outputs with a `number` greater than one, the required number of envelopes or inputs are appended sequentially, as shown in Figure 3.19.

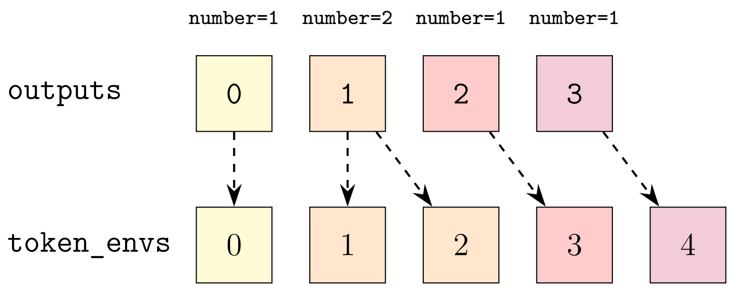


Figure 3.19: Ordering convention of `outputs` and `token_envs`.

Request

```
interface PayRequest {
  // other fields omitted for brevity
  wallet_data?: PayWalletData;
  tokens?: TokenPaySig[];
}

interface PayWalletData {
  choice_index?: Integer;
  tokens_evs?: TokenEnvelope[];
}

interface TokenPaySig {
  token_sig: EddsaSignature;
  token_pub: EddsaPublicKey;
  ub_sig: UnblindedSignature;
}

type TokenEnvelope = RsaTokenEnvelope | CsTokenEnvelope;

interface RsaTokenEnvelope {
  cipher: "RSA";
  rsa_blinded_pub: BlindedRsaSignature;
}

interface CsTokenEnvelope {
  cipher: "CS";
  cs_nonce: CsNonce;
  cs_blinded_c0: Cs25519Scalar;
  cs_blinded_c1: Cs25519Scalar;
}

type UnblindedSignature = UnblindedRsaSignature | UnblindedCsSignature;

interface UnblindedRsaSignature {
  cipher: "RSA";
  rsa_signature: RsaSignature;
}

interface UnblindedCsSignature {
  cipher: "CS";

  cs_signature_r: Cs25519Point;
  cs_signature_s: Cs25519Scalar;
}
```

Response

```
interface PaymentResponse {
  // other fields omitted for brevity
  token_sigs?: SignedTokenEnvelope[];
}

interface SignedTokenEnvelope {
  blind_sig: TokenIssueBlindSig;
}

type TokenIssueBlindSig = RSATokenIssueBlindSig | CSTokenIssueBlindSig;

interface RSATokenIssueBlindSig {
  cipher: "RSA";

  blinded_rsa_signature: BlindedRsaSignature;
}

interface CSTokenIssueBlindSig {
  type: "CS";

  b: Integer;
  s: Cs25519Scalar;
}
```

Example API interactions

The below output illustrates how a client could interact with the merchant backend RESTful API to purchase a subscription. The examples use the CURL[5] command line tool to make HTTP request, but any other tool or programming language could be used.

For the sake of these examples, it is assumed that the merchant backend is accessible at `https://backend.demo.taler.net` and that `secret-token:sandbox` is configured as a valid authentication token for the private API. Additionally, it is assumed that a trusted exchange has been configured for the `KUDOS` currency.

The command shown in Figure 3.20 creates a new subscription token family that is valid between August 1, 2024 00:00 GMT and August 1, 2025 00:00 GMT. Issued tokens of this token family are valid for 30 days. The request returns a HTTP status code 204 (no content).

[5] `https://curl.se/`

```
curl --url https://backend.demo.taler.net/private/tokenfamilies \
  --header 'Content-Type: application/json' \
  --header 'Authorization: Bearer secret-token:sandbox' \
  --data '{
  "slug": "monthly-sub-2024",
  "kind": "subscription",
  "name": "Monthly Subscription 2024",
  "description": "This is a test subscription.",
  "valid_after": { "t_s": 1722470400 },
  "valid_before": { "t_s": 1754006400 },
  "duration": { "d_us": 2592000000000 },
  "rounding" { "d_us": 2592000000000 }
}'
```

Figure 3.20: Creating a subscription token family.

Once a token family has been set up, an order is created that issues a token of this family. This process is illustrated in Figure 3.21. This order enables the purchase of a subscription. To reduce the complexity of the payment request for the purposes of this example, the order has a price of zero. This means that no coins have to be provided during payment. The `number` and `valid_after` fields of the output are not explicitly specified in the request, thus the defaults will be used for these fields. Consequently, the resulting order will issue one token that will be valid immediately.

```
curl --url https://backend.demo.taler.net/private/orders \
  --header 'Content-Type: application/json' \
  --header 'Authorization: Bearer secret-token:sandbox' \
  --data '{
  "order": {
    "version": 1,
    "order_id": "buy-subscription",
    "summary": "Buy a monthly subscription.",
    "fulfillment_message": "Payment successful.",
    "choices": [
      {
        "price": "KUDOS:0",
        "outputs": [
          {
            "kind": "token",
            "token_family_slug": "monthly-sub-2024",
          }
        ]
      }
    ]
  }
}'
{
  "order_id": "buy-subscription",
  "token": "HMBM97CQZ05SSJ8DAWKGXS0QR8"
}
```

Figure 3.21: Creating an order that issues a token.

The request depicted in Figure 3.21 returned a claim token[6] that is used to claim an order. As illustrated in Figure 3.22, a wallet-generated nonce is passed along with the claim request. The response, depicted in Figure 3.23, includes the complete contract terms with the token issue public key of the subscription output token. It should be noted that response fields irrelevant to this example have been omitted for brevity.

```
curl --url https://backend.demo.taler.net/orders/buy-subscription/claim \
  --header 'Content-Type: application/json' \
  --data '{
  "nonce": "my-wallet-nonce",
  "token": "HMBM97CQZ05SSJ8DAWKGXS0QR8"
}'
```

Figure 3.22: Request for claiming an order

[6]This is a simple string generated upon order creation and unrelated to subscription or discount tokens.

```
{
  "contract_terms": {
    "version": 1,
    "summary": "Buy a monthly subscription.",
    "fulfillment_message": "Payment successful.",
    "choices": [
      {
        "price": "KUDOS:0",
        "inputs": [],
        "outputs": [
          {
            "kind": "token",
            "token_family_slug": "monthly-sub-2024",
            "number": 1,
            "valid_after": {
              "t_s": 1775001600
            }
          }
        ]
      }
    ],
    "token_families": {
      "monthly-sub-2024": {
        "name": "Monthly Subscription 2024",
        "description": "This is a test subscription.",
        "keys": [
          {
            "cipher": "RSA",
            "rsa_pub": "080000Z48AS8YQWPB9SS4...",
            "valid_after": {
              "t_s": 1775001600
            },
            "valid_before": {
              "t_s": 1777593600
            }
          }
        ],
        "critical": true
      }
    },
    "nonce": "my-wallet-nonce"
  }
```

Figure 3.23: Claim order response (some fields are omitted for brevity)

Once the order has been claimed, it is ready for payment. As the order outputs a token, the wallet must provide a token envelope. Additionally, the wallet must indicate the selected choice by setting the `choice_index` to the respective index. As the example order only has a single choice, `choice_index` is set to `0` in the pay request shown in Figure 3.24. The array of `coins` is left empty, as the selected choice has a price of zero. The blinded token use public key is generated by the client (not shown in this example) and included in a token envelope in the respective array of the `wallet_data` object of the request.

```
curl --url https://backend.demo.taler.net/orders/buy-subscription/pay \
  --header 'Content-Type: application/json' \
  --data '{
  "coins": [],
  "wallet_data": {
    "choice_index": 0,
    "tokens_evs": [
      {
        "token_ev": {
          "cipher": "RSA",
          "rsa_blinded_planchet": "AFF85QZA6N..."
        }
      }
    ]
  }
}'
{
  "token_sigs": [
    {
      "blind_sig": {
        "cipher": "RSA",
        "blinded_rsa_signature": "C5A41X66S..."
      }
    }
  ],
  "sig": "9F0NXFTGNFM..."
}
```

Figure 3.24: Paying for an order that issues a token.

In the response of Figure 3.24 the merchant backend issues a blinded token in the form of a blinded RSA signature. To use this token on an order, an order must be created to accept using such a token. As illustrated in the request depicted in Figure 3.25, the order has two choices how it can be paid. The client may choose to pay by regular coins or by using a subscription token.

```
curl --url https://backend.demo.taler.net/private/orders \
  --header 'Authorization: Bearer secret-token:sandbox' \
  --header 'Content-Type: application/json' \
  --data '{
  "order": {
    "version": 1,
    "order_id": "news-article",
    "summary": "Get access to a news article.",
    "fulfillment_message": "Payment successful.",
    "choices": [
      {
        "price": "KUDOS:5",
        "outputs": [
          {
            "kind": "token",
            "token_family_slug": "monthly-sub-2024"
          }
        ]
      },
      {
        "price": "KUDOS:0",
        "inputs": [
          {
            "kind": "token",
            "token_family_slug": "monthly-sub-2024"
          }
        ],
        "outputs": [
          {
            "kind": "token",
            "token_family_slug": "monthly-sub-2024"
          }
        ]
      }
    ]
  }
}'
{
  "order_id": "buy-subscription",
  "token": "HMBM97CQZ05SSJ8DAWKGXS0QR8"
}
```

Figure 3.25: Creating an order that allows to use a subscription token.

To use the token issued in the response of Figure 3.24, the client is required to select the corresponding choice from the order created in Figure 3.25. The unblinded token issue signature, the token public key and the token use signature must be included in the `tokens` field of the pay request, shown in Figure 3.26. As the order issues a new token, the client is required to include a corresponding token envelope.

```
curl --url https://backend.demo.taler.net/orders/news-article/pay \
  --header 'Content-Type: application/json' \
  --data '{
  "coins": [],
  "tokens": [
    {
      "token_sig": "Q8JSCT2B...",
      "token_pub": "9N0341PD...",
      "ub_sig": {
        "cipher": "RSA",
        "rsa_signature": "42BVNNWRD..."
      }
    }
  ],
  "wallet_data": {
    "choice_index": 1,
    "tokens_evs": [
      {
        "token_ev": {
          "cipher": "RSA",
          "rsa_blinded_planchet": "G1QYGXPM9S0..."
        }
      }
    ]
  }
}'
{
  "token_sigs": [
    {
      "blind_sig": {
        "cipher": "RSA",
        "blinded_rsa_signature": "BF4Q21S96..."
      }
    }
  ],
  "sig": "5510NBZK..."
}
```

Figure 3.26: Paying for an order by using a subscription token.

3.7. Database

The existing schema of the merchant PostgreSQL database was extended by four tables: token families, their signing keys, issued tokens, and used tokens (see Figure 3.27).

Each row in `merchant_token_families` represents a token family. The merchant maintains a record of the number of tokens that have been issued and used for each token family. In order to facilitate faster and more convenient lookups, these counters are stored directly in this table, rather than being calculated by a join operation.

One token family may have none, one, or many signing keys stored in the `merchant_token_family_keys` table. Keys are stored in a binary format, where the first few bytes indicate the cipher of the key. In addition to the public and private key, the hash of the public key is stored and indexed in the database for performant lookups. The validity timeframe of a key is denoted by `valid_after` and `valid_before`.

In order to prevent double spending of tokens, merchants will maintain a record of used tokens in the `merchant_used_tokens` table. This table contains the EdDSA public key of a token, a reference to its signing key, the token issue signature, and the token use signature done using the corresponding private key of the token.

Furthermore, the merchant uses the `merchant_issued_tokens` table to store the blind signatures created during the issuance process. This table is useful for ensuring the idempotency of the payment endpoint, which means that the same request leads to the same response, even when it is replayed. Issued tokens are stored in this table to ensure they can be returned if a pay request is repeated. This mechanism also serves to assist the wallet in recovering from an unexpected interruption to a request.

Once a token issue key has expired, all issued and used tokens of that key can be removed safely, as they are no longer valid. This process ensures that the storage requirements of the database do not grow indefinitely.

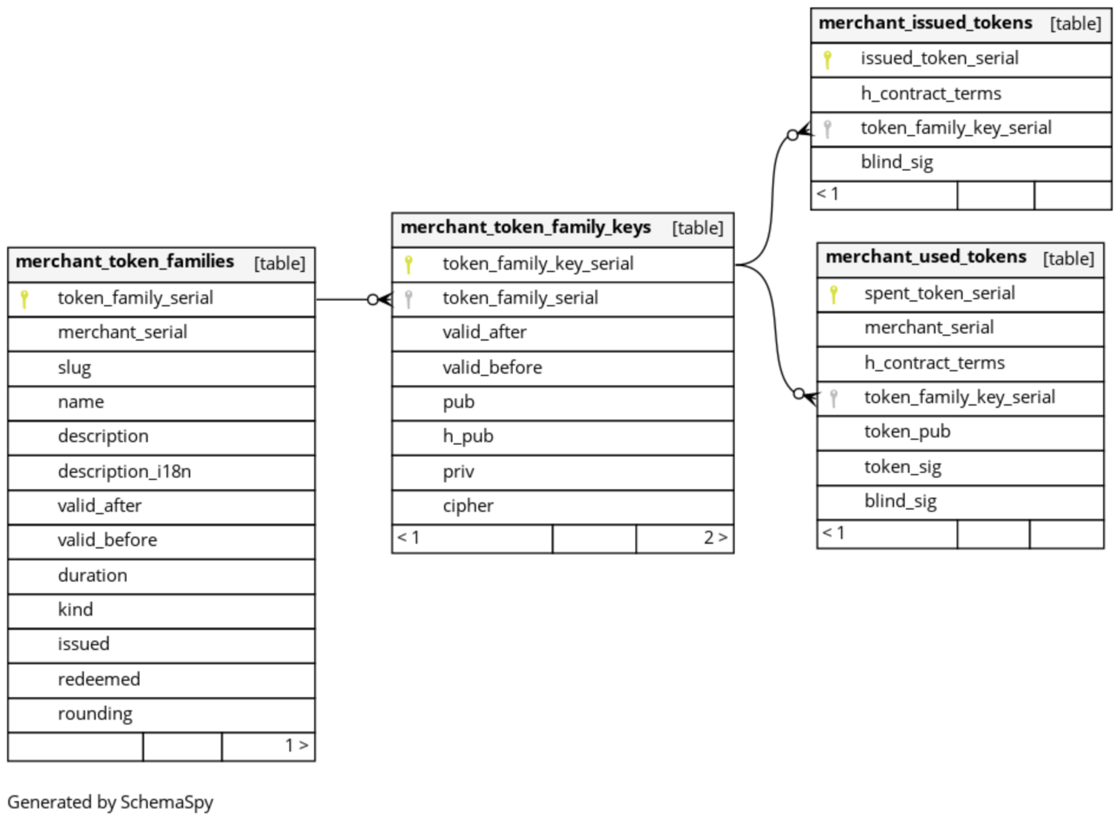


Figure 3.27: Tables added to merchant database.

4. Implementation

The proposed solution is primarily written in the C programming language. This is due to its integration with the GNU Taler project, which is predominantly written in C. The web GUIs are written in TypeScript, utilising Preact[1], a lightweight virtual Document Object Model (DOM) abstraction compatible with the API of the popular React[2] library.

The GNU Taler project is comprised of multiple repositories, each containing multiple applications, packages and libraries. Figure 4.1 provides a high-level overview of all parts of the project that required modification to implement token-based subscriptions and discounts.

4.1. Token family management

As previously described in Subsection 3.6, the merchant backend exposes a private API that allows authorized clients to configure token families. In a typical setup, one of these clients is the merchant back office SPA.

The first step in this process is to implement the new endpoints in the merchant backend. This includes adding the endpoints to the HTTP router, writing the corresponding handler functions, modifying the database, and extending the database package with the relevant queries. In a subsequent phase, the web Graphical User Interface (GUI) must be extended to permit merchant staff to configure token families. This configuration step should be as user-friendly as possible, requiring no technical expertise. Consequently, users of the merchant back office SPA should never see any keys or configure which cipher is used. The technical configuration is performed by the administrator of the merchant backend in the respective configuration files.

The merchant backend depends on the GNU Libmicrohttpd[3] library to implement a web server. In addition to serving the RESTful API, it is also used to serve the merchant back office SPA under the `/webui` path.

The merchant backend defines all paths of its API in the file `src/backend/taler-merchant-httpd.c`. The handlers are divided into thee groups: management han-

[1]`https://preactjs.com/`
[2]`https://react.dev/`
[3]`https://www.gnu.org/software/libmicrohttpd/`

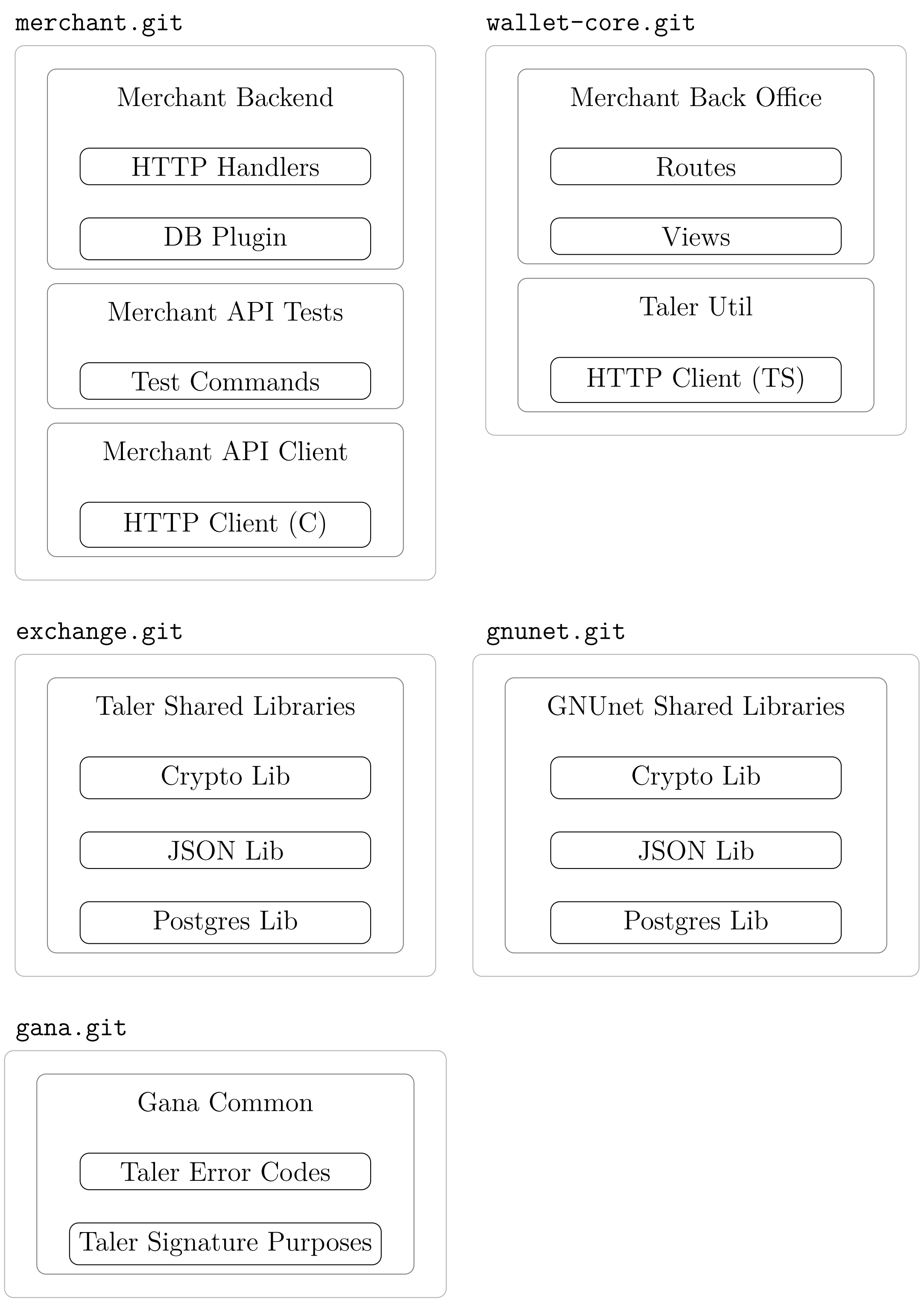


Figure 4.1: Overview of relevant repositories, components and packages.

dlers, private handlers, and public handlers. Since the token family API will be private, this is where we can add all paths listed in the API design specification (see Figure 4.2).

A request handler is defined as an instance of `struct TMH_RequestHandler`. It contains all the necessary information for the server to build the path, extract the optional ID parameter, and route HTTP requests based on the combination of path and method to the defined handler function. The definition of the URL path is divided into two variables: `url_prefix` and `url_suffix`. These variables are separated by the ID segment. Since the token family API does not have any endpoints with a suffix after the ID segment, `url_suffix` is not used in the code snippet in Figure 4.2. In the context of the token family API, the ID segment serves as the slug parameter utilized in GET, DELETE and PATCH requests.

```
{
  .url_prefix = "/tokenfamilies",
  .method = MHD_HTTP_METHOD_GET,
  .handler = &TMH_private_get_tokenfamilies
},
{
  .url_prefix = "/tokenfamilies",
  .method = MHD_HTTP_METHOD_POST,
  .handler = &TMH_private_post_token_families
},
{
  .url_prefix = "/tokenfamilies/",
  .method = MHD_HTTP_METHOD_GET,
  .have_id_segment = true,
  .handler = &TMH_private_get_tokenfamilies_SLUG
},
{
  .url_prefix = "/tokenfamilies/",
  .method = MHD_HTTP_METHOD_DELETE,
  .have_id_segment = true,
  .handler = &TMH_private_delete_token_families_SLUG
},
{
  .url_prefix = "/tokenfamilies/",
  .method = MHD_HTTP_METHOD_PATCH,
  .have_id_segment = true,
  .handler = &TMH_private_patch_token_family_SLUG,
},
```

Figure 4.2: HTTP handlers defined in `taler-merchant-httpd.c`.

Every request handler is associated with a unique file within the `src/backend/` directory of the merchant backend codebase. For instance, the file `taler-merchant-httpd_private-get-token-families.c` contains the handler for the GET /token-

```
struct GNUNET_PQ_QueryParam params[] = {
  GNUNET_PQ_query_param_string (instance_id),
  GNUNET_PQ_query_param_end
};
enum GNUNET_DB_QueryStatus qs;
check_connection (pg);
PREPARE (pg,
         "lookup_token_families",
         "SELECT slug, name, valid_after, valid_before, kind"
         "FROM merchant_token_families"
         "JOIN merchant_instances USING (merchant_serial)"
         " WHERE merchant_instances.merchant_id=$1");
qs = GNUNET_PQ_eval_prepared_multi_select (pg->conn,
                                            "lookup_token_families",
                                            params,
                                            &lookup_token_families_cb,
                                            &context);
```

Figure 4.3: Usage of GNUnet PQ library for listing token families.

families request, along with its corresponding `.h` header file, which contains the function signature of the handler.

In the token family management API, all endpoints follow a similar structure for processing incoming requests. First, they sanitize and validate the incoming user-provided data. Then, they send a SQL query to the database and handle the result accordingly. For all user-provided parameters prepared SQL statements are used to mitigate injection-related security risks. If the database query returns an unexpected result, the web server will return an error to the client. In the event that the database query is successful, the data returned by the query is parsed into the corresponding C data structures and returned to the calling function.

All database interactions are defined in their own files, located in the `/src/backenddb/` directory. To interact with the database, the merchant backend depends on the GNUnet PQ library[4]. The code snippet in Figure 4.3 illustrates how the GNUnet PG library is used to execute queries with input parameters. It initializes an array of parameters, which it then passes into the prepared statement with placeholders for each. To define the parameters, it makes use of helper functions for each data type. In Figure 4.3 the helper function `GNUNET_PQ_query_param_string` is used to interpret the provided parameter as a string.

The result of the query will be parsed into a C data structure by the callback function `lookup_token_families_cb`, which will be called once the database request returns. In a manner similar to how it handles input parameters, the GNUnet PQ library also provides helper functions to parse common data types from SQL back into C code. For further details, please refer to the code snippet in 4.4.

[4]`https://git.gnunet.org/gnunet.git/tree/src/include/gnunet_pq_lib.h`

```c
char *slug;
char *name;
char *kind;
struct GNUNET_TIME_Timestamp valid_after;
struct GNUNET_TIME_Timestamp valid_before;
struct GNUNET_PQ_ResultSpec rs[] = {
  GNUNET_PQ_result_spec_string ("slug",
                                &slug),
  GNUNET_PQ_result_spec_string ("name",
                                &name),
  GNUNET_PQ_result_spec_timestamp ("valid_after",
                                   &valid_after),
  GNUNET_PQ_result_spec_timestamp ("valid_before",
                                   &valid_before),
  GNUNET_PQ_result_spec_string ("kind",
                                   &kind),
  GNUNET_PQ_result_spec_end
};
/* Error handling omitted for brevity */
GNUNET_PQ_extract_result (result, rs, i))
```

Figure 4.4: Using GNUnet PQ library to parse SQL results into C data structures.

Once the query has been executed successfully and the data has been parsed, the HTTP handler function is ready to begin constructing the response to be sent to the client. Some endpoints do not return any data in the event of a successful completion (HTTP status code 204 No Content), while others return a JSON body (HTTP status code 200 OK). In order to parse and serialize JSON, the merchant backend depends on the GNUnet JSON library[5], which itself depends on the popular jansson[6] library. The API of the GNUnet JSON library is analogous to that of GNUnet PQ, but instead of SQL it parses and serializes JSON. Figure 4.5 illustrates its use in conjunction with the `json_array_append_new` function from the jansson library. Following successful serialization, the server transmits the JSON response body to the client. Depending on the handler function, it is necessary to free some dynamically allocated memory. This marks the conclusion of the process of handling an HTTP request for the merchant backend server.

The token family management API is consumed by the merchant back office SPA. The HTTP client class `TalerMerchantInstanceHttpClient` is responsible for making HTTP requests, adding optional authentication headers, and handling potential errors. The error handling logic is capable of differentiating between known error response codes (as listed in the merchant API documentation[7]) and unexpected errors. This is beneficial, as these errors have to be handled in different ways.

[5]https://git.gnunet.org/gnunet.git/tree/src/include/gnunet_json_lib.h
[6]https://github.com/akheron/jansson
[7]https://docs.taler.net/core/api-merchant.html

```
json_array_append_new (
  pa,
  GNUNET_JSON_PACK (
    GNUNET_JSON_pack_string ("slug", slug),
    GNUNET_JSON_pack_string ("name", name),
    GNUNET_JSON_pack_timestamp ("valid_after", valid_after),
    GNUNET_JSON_pack_timestamp ("valid_before", valid_before),
    GNUNET_JSON_pack_string ("kind", kind)));
```

Figure 4.5: Serializing a JSON object and appending it to an array.

In order to facilitate the sharing of the HTTP client code across multiple TypeScript projects, it is located in a shared package that is imported by the main application. In the application, the listing of all token families and inspection of a token family requests of the HTTP client are wrapped in Stale-While-Revalidate (SWR) hooks[8], a React data-fetching pattern that uses a cache invalidation strategy specified in [25]. The use of SWR hooks enable the creation of a more responsive user experience, particularly for devices with weak or unreliable internet connections.

The inherit nature of a SPA is to consist of a single `index.html` file. However, this raises the question of how the browser navigates to different pages within the application. Instead of using conventional links that point to different HTML files on the web server, the merchant back office SPA uses a process called hash-based client-side routing. Upon clicking a link to another page, JavaScript code is executed, modifying the fragment (as shown in Figure 4.6) of the current URL. Since the path of the current URL remains unchanged, this modification does not resulting in a request being sent to the server. Instead, it enables the Preact router[9] to react and load the requested page.

$$\underbrace{\text{http://}}_{\text{Scheme}}\underbrace{\text{www.example.com}}_{\text{Domain}}\underbrace{\text{/path/to/resource}}_{\text{Path}}\underbrace{\text{\#section}}_{\text{Fragment}}$$

Figure 4.6: Components of an URL.

All routes of the application are defined in the file `/src/Routing.tsx`. The `.tsx` file extension stands for TypeScript Syntax Extension. It allows developers to write HTML-like markup directly in TypeScript code that is transpiled into native JavaScript code during the build process.

For managing token families, the creation, listing and updating pages were added to the application. The list page retrieves all available token families from the backend and presents them in a table view. This allows users to update and delete token families without leaving the page. A prominent button at the top will navigate to the creation page, which will render a form to input all necessary information for configuring a new token family. Guidance texts explain the different fields to users.

[8]`https://swr.vercel.app/`

[9]`https://www.npmjs.com/package/preact-router`

After a token family is created, some information can be changed on the update page. It should be noted that this does not include information such as the token family kind or the slug of the token family. Instead of updating these fields, users are encouraged to delete and create a new token family.

4.2. Order creation

In order to implement the creation of orders in the format specified in Section 3.2, it is necessary to modify the corresponding HTTP handler. This handler must parse, validate and store the optionally provided array of choices within an order. Furthermore, the merchant backend must also check if there is a valid key pair for each referenced token family and generate a new one if not.

Given the complexity of this process, the create order HTTP handler function is divided into multiple phases. Each phase determines the subsequent phase based on the incoming request, much like a finite-state machine.

As illustrated in Figure 4.7, most phases can directly transition into the error finish state. This state will complete the processing of the request and return the captured error to the client. Conversly, if no error occurs, a request will transition through all phases, ultimatively arriving at the success phase. This phase will return a HTTP status code of 200 OK and a JSON body containing the order ID and the token used to claim the order.

As is common for most APIs in the GNU Taler project, the create order endpoint is idempotent. This means that even if the same request is made multiple times, the outcome will be the same as if it was only made once. Idempotency allows a client to replay a request if it was unable to capture the response of the initial request. This can occur due to network outages, unexpected crashed, forced termination, or other unforeseen errors. To enable the order creation endpoint to be idempotent, the backend queries the database, checking if there is already an order with the same order ID. If an order with the same ID exists, it compares the hash of the request body to that of the original order in the database. If they match, the request is idempotent and the backend will return the order stored in the database. If the hash of the request body does not match, the backend will return an error message with the HTTP status code 409 Conflict, indicating that an order with the specified ID already exists.

During the parse choices phase, the merchant backend will construct a list of all referenced token families and the matching token issue public keys. This is achieved by iterating over the inputs and outputs of all choices. For each token family that is not already present in the list, the backend will fetch its metadata, as well as the public key matching the provided valid after time. If no matching key was found in the database, the backend will generate a new key pair and insert it into the database.

To generate a new key pair, the merchant backend depends on the GNUnet crypto

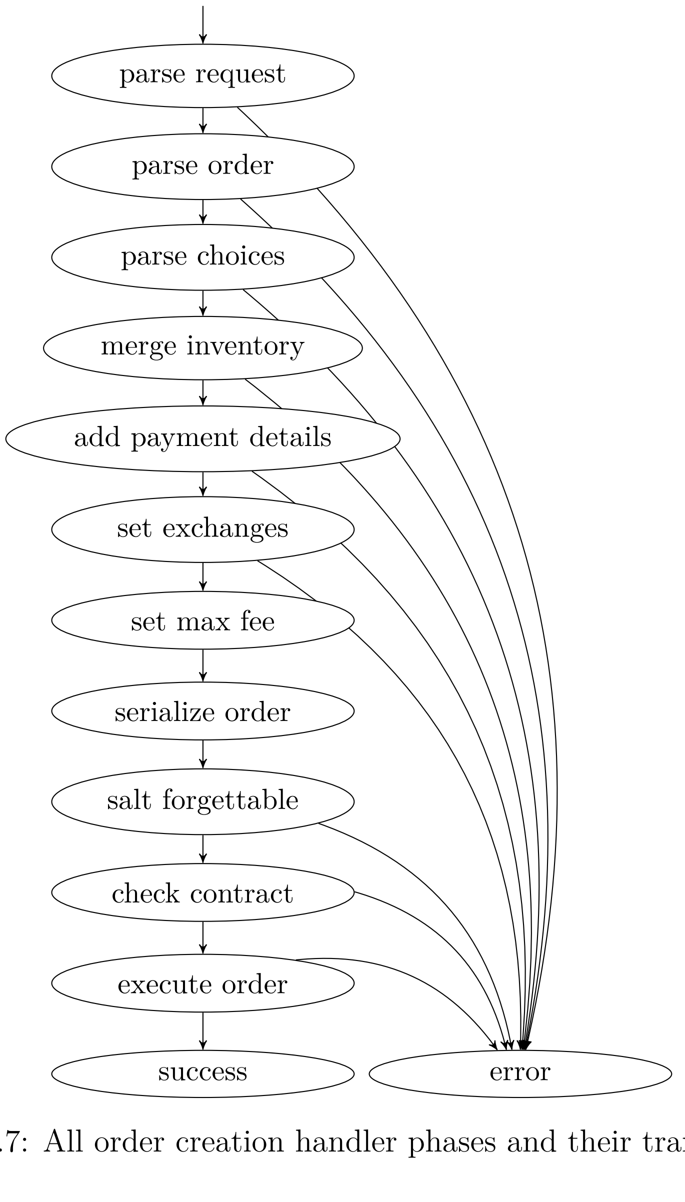


Figure 4.7: All order creation handler phases and their transitions.

library[10]. The GNUnet crypto library offers an abstracted blind signing API that supports RSA as well as Clause Schnorr blind signature schemes. Under the hood, the GNUnet crypto library uses Libgcrypt[11] for RSA operations and Libsodium[12] for the generation of Clause Schnorr key pairs.

When generating a new key pair, the validity start date is derived from the current system time of the backend. The current time is rounded according to the logic specified in Section 3.1. Then, the configured token duration of the token family is added to the rounded start date, resulting in the end date.

Once the key pair has been generated and its validity timeframe has been derived, it is inserted into the database. The keys are encoded in a binary format, where the first four bytes indicate whether it is an RSA or a Clause Schnorr key. Alongside the public and private keys, the hash of the public key is also stored in the database, allowing for fast key lookups. This step concludes the token issue key generation process.

The merchant backend stores orders as JSON objects in the database, thereby freezing the system state at the time of creation. Information such as metadata of token families is copied into the JSON object. This approach ensures that the contract terms remain unchanged, even if merchant staff later modify the token family description. The structure of the JSON order in the database follows that specified in Section 3.2.

All relevant token families are stored in a map on the order JSON, with the key of the map being the token family slug. In addition to name and description, each token family has a boolean field marking it as critical or not. This field indicates whether a wallet must understand a given token in order to process contracts that utilize or issue it. For instance, subscription tokens are considered critical, whereas discount tokens are not. A wallet can readily ignore a discount token that is issued, whereas a subscription token represents a critical component of a given process. Each token family also comprises an array of keys relevant to the given order. Besides the token issue public key, each key features a field indicating the cipher used and the validity timeframe of tokens signed by this issue key.

Additionally, the order JSON object comprises an array of choices. These choices are presented to the wallet during the payment process prompting the customer to select one. Each order can contain multiple inputs and outputs of different types. For the proposed solution, only token-based input and outputs are relevant. Each input or output consists of the token family slug, valid after date and an optional number field. The slug can be used for constant-time lookup of metadata of the token family, while the valid after date matches with one of the keys on this family. The optional number property is used to specify how many tokens are required or issued. It defaults to one and input or output with a number field equal to zero are ignored during the order creation process.

[10]`https://git.gnunet.org/gnunet.git/tree/src/include/gnunet_crypto_lib.h`

[11]`https://gnupg.org/software/libgcrypt/index.html`

[12]`https://libsodium.gitbook.io/doc`

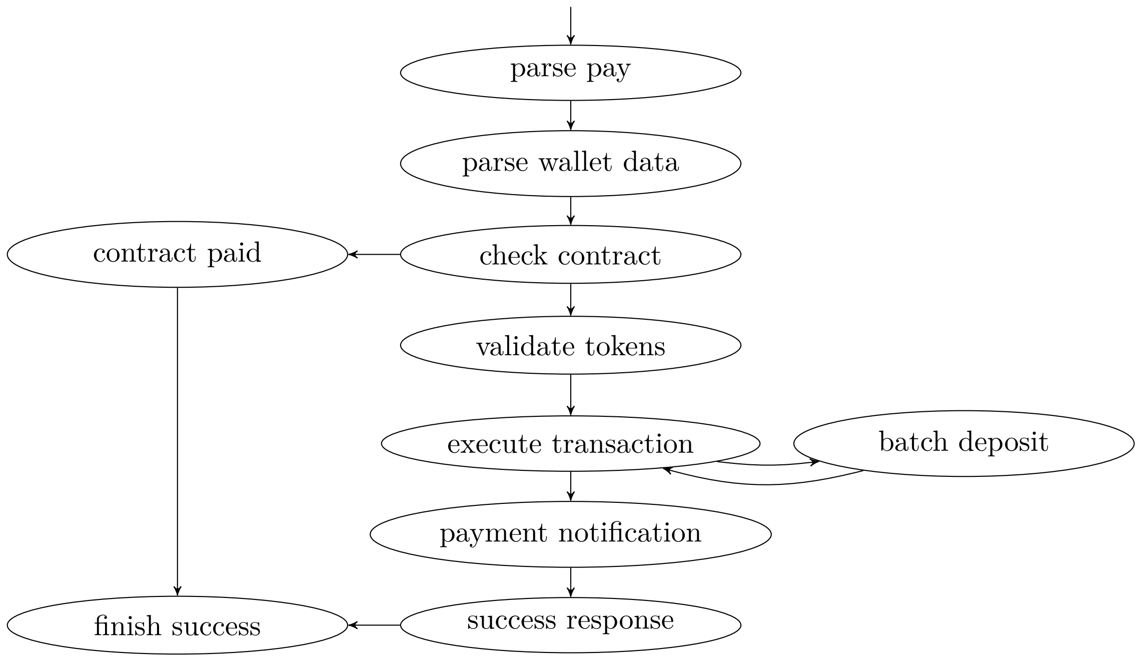


Figure 4.8: Payment processing handler phases and their transitions. All states, expect 'finish success', can also transition into a final error state.

4.3. Payment processing

The merchant backend exposes a pay endpoint as part of its public API, allowing wallets to pay for an order. When processing the payment for a new v1 order as specified in Section 3.2, the merchant backend must additionally validate input tokens and blindly sign token envelopes for output tokens. Similarly to the order creation handler, the payment processing handler is also divided into multiple phases, as shown in Figure 4.8. With the exception of 'finish success', all phases shown in the figure can also transition into an error state.

As specified in Subsection 3.6, the pay request is extended with two additional array fields: input tokens required by the order and blinded token envelopes for output tokens. Additionally, the wallet passes the index of the selected choice in the wallet data object. The hash of the wallet data object is also part of the token use request. This request is signed by the wallet with the private key of a token when it is used on an order. This unambiguously ties the usage of a token to a selected contract choice and commits the wallet to the token envelopes provided. A similar signature, including the same wallet data object, is done when spending a coin. Therefore a coin is also bound to a selected choice, eliminating any ambiguities which choice of an order was selected or which token envelopes were provided.

The merchant backend performs input tokens validation as early as possible in the payment process. This allows it to return any errors and avoid further processing of the request. As seen in Figure 4.8, the 'validate tokens' phase is initiated after the request has been parsed and the contract terms have been loaded from the database. Most importantly, the token validation is completed before the coins of an order are deposited at the exchange during the 'batch deposit' phase. This reduces the network traffic to the centralized exchange service, as invalid requests do not trigger a request from the merchant to the exchange. In the validate token phase, input tokens provided by the wallet are validated. This phase is omitted for v0 orders. During the validation process, the backend iterates over all input tokens of the selected choice. It employs the ordering convention described in Subsection 3.6 to identify the matching token family and issue key for an input. This enables the identification of the keys in constant time, without having to loop over all token families and their keys. The token issue public key is then used to verify the token issue signature provided by the wallet. To validate the signature, the backend passes the hash of the provided token use public key and the token issue public key into the cryptographic verify function of the configured signature scheme. In the event of a verification failure, the backend will cease processing any further input tokens and return the error to the client. If it succeeds, it will continue by validating the token use signature. The token use signature is an EdDSA signature generated by the wallet using the token use private key. The signed message is the token use request, as specified in Figure 3.15. The merchant backend has all information required to construct the token use request, including the token use public key to validate it. In the event that the signature does not verify, the backend will promptly cease processing any further input tokens and return a detailed error message to the client. If no error was encountered so far, the backend proceeds to signing the token envelopes.

A token envelope contains a blinded hash of the token use public key that was generated by the wallet. This blinded message is signed with the token issue private key in a process called *token issuance*. Analogous to input tokens, the merchant backend uses the same ordering convention, as described in Subsection 3.6, to identify the key corresponding to an envelope. The wallet must provide the specified number of envelopes for each critical output token, with no more than the total number of output tokens for the selected choice. If all envelopes were signed successfully, the backend stores the signed envelopes in the database. This allows the backend to return the same envelopes on an idempotent pay request. Furthermore, this ensures that the token envelopes are signed prior to the request being made to the exchange, thus guaranteeing that the signed token envelopes are available once the exchange has completed the processing of the deposited coins. This design decision eliminates the creation of a tedious error chain that would occur if tokens were signed after the deposit request was made to the exchange. In that case, failure to sign a token (e.g., because the issue private key has been deleted in the meantime) would require the merchant to refund the payment before returning the error to the wallet.

4.4. Order details

It is necessary for the merchant backend to maintain a record of which choice was paid for v1 orders. This information is stored in a new `choice_index` column of the contract terms table. The contract terms table contains data on orders that have been paid. This field is also returned as part of the private order details API, as specified in Subsection 3.6.

The GNU Taler project employs a database migration management framework that ensures migrations are applied exactly once. To insert the new field into the existing table, a SQL migration was created. Figure 4.9 contains a snippet of the migration. A nullable two-byte integer field was added, as the choice index will always be a relatively small integer. For v0 orders, the column will be set to null.

```sql
ALTER TABLE merchant_contract_terms
  ADD COLUMN choice_index INT2 DEFAULT NULL;

COMMENT ON COLUMN merchant_contract_terms.choice_index
  IS 'Index of selected choice. Refers to the `choices` array in the
          contract terms. NULL for contracts without choices.';
```

Figure 4.9: Snippet of SQL migration to add choice index column.

After adding the column, the insert and select queries used by the database plugin had to be modified to set and retrieve the value. Given that integers are not inherently nullable in the C programming language, rows with a choice index set to null result in the corresponding variable within the C code being set to the special value of `-1`. Consequently, the order details HTTP handler will only add the choice index field to the JSON response if it is greater or equal to zero. Therefore, only responses for v1 orders will include this new field, while responses for older orders will remain unchanged. This ensures the backward compatibility of the order details API.

4.5. Merchant API tests

Automated tests represent a crucial component in ensuring that implemented code functions as intended. To test the implemented solution, test cases were defined and integrated into the existing merchant test suite.

The merchant test suite interacts with the merchant via its private and public APIs. There are two categories of tests. The first are simple shell scripts that use the CURL[13] command line tool to make HTTP requests and validate their responses. These tests may be considered as smoke tests[14], which are designed to identify and capture major errors during development. They are primarily positive tests, which test the "happy path" and are expected to succeed. The second category of tests are more complex and are written in C. These test cases emulate more involved flows and also include negative tests, which are tests that are expected to return an error. One example of such a negative test is the rejection of expired input tokens during the payment process.

[13]`https://curl.se/`

[14]`https://en.wikipedia.org/wiki/Smoke_testing_(software)`

The test cases for the extended merchant backend API have been designed to cover all core features of the system. They also serve as the reference implementation for wallets, since EdDSA key generation, blinding and signing was implemented as part of the test suite code written in C. The following is a list of all test cases implemented during the development of the system. It should be noted that later test cases depend on the tests that came before them in the list.

1. **Creating a token family.** This test case serves to verify the functionality of the token family creation process. It initiates a HTTP POST request to the `/tokenfamilies` endpoint of the private API and awaits a response status code of 204 No Content. This test case also establishes the foundation for subsequent tests, as they all of them interact with tokens issued by a token family.

2. **Buying a subscription.** This test case requires that a token family of type 'subscription' has already been set up. It uses the private API of the merchant backend to create an order with a non-zero price and one output token of the previously configured token family. This order represents the offer of buying a subscription token. After the order is created successfully, the test suite uses the public API to pay for it. Prior to issuing the HTTP request to the `/ORDER_ID/pay` endpoint, the test suite must prepare the token envelopes. This is accomplished by generating an EdDSA key pair[15] and blinding the hash of the public key. The test case will succeed if the response has the status code 200 OK. Upon receiving a successful response, the test suite will unblind the token and make it accessible to subsequent test cases within the execution chain.

3. **Using a subscription.** Prior to utilizing a subscription, the test suite must first possess a valid subscription token. This subscription token was acquired in the previous test case. The test case can then use the merchant backend private API to set up an order with a price of zero, which requires an input token and issues an output token of the same token family. Such an order effectively models the process of using a subscription. To fulfill this order, the test suite must prepare the token envelope and sign a token use request with the token use private key. The resulting signature, the prepared envelope, and the input token are then transmitted as part of the pay request to the merchant backend. The test suite expects this request to be responded with a status code of 200 OK.

[15]the token use key

5. Future work and discussion

This chapter discusses the work undertaken in this thesis and identifies some potential topics for future research.

5.1. Backup-friendly wallet implementation

The wallet implementation is essential for the proposed solution to work end-to-end. Therefore, this is the most pressing topic for future work. Initially the wallet implementation was planed as part of this thesis, but it got clear in the early stages of the implementation that the limited timely resources would not suffice for implementing token-based subscriptions and discounts in the GNU Taler wallets.

The protocol for token-based subscriptions and discounts between merchant backend and wallet is specified and documented. Additionally, the interactions of the wallet are implemented in a simplified manner in the merchant backend tests in Section 4.5. This code can serve as a reference for the integration of tokens into the wallet, with the hope of making that part of the implementation straightforward.

More interesting is *how* the wallet will generate or derive new tokens. When using a subscription, the wallet could derive the private key of the replacement token, along with the blinding secret used, from a *master secret* distinct for this subscription combined with a serial number n. This approach could facilitate convenient backups of subscriptions in wallets. In the event of a recovery from a backup, the wallet should be able to obtain the blinded issue signature of a token. This can be achieved either by the wallet being able to reconstruct the idempotent pay request or by the merchant backend exposing a new endpoint where the wallet can request the blind signature. However, this approach has the disadvantage of making subscriptions easily sharable between multiple users. Consequently, if an individual has an old subscription token, they can use the backup process to obtain the latest, and therefore valid, token in the chain.

As the implications of such a derivation mechanism must be studied and evaluated in detail, this could prove an interesting topic for future research, potentially as a thesis for a student.

5.2. ASS authority

As previously stated in Section 3.1, the size of the anonymity set is critical to guarantee the anonymity of customers. A dishonest merchant could generate a new token issue key for each individual customer, thereby enabling the merchant to link the actions of said customer. In order to prevent such misuse, an anonymity set size authority is proposed. This authority would provide customers with transparent information regarding the anonymity set size of a subscription, enabling them to assess their comfort level in continuing to use it. Such an authority would reduce the level of trust that customers must place in merchants.

5.3. Refund tokens

The GNU Taler payment system is designed in such a way that orders can be refunded while retaining the unlinkability of refunded coins. To refund input tokens, the merchant could reissue the tokens of the same token family to the customer. This would require the customer to present the unblinded public key of all output tokens issued in the to-be refunded transaction. This allows the merchant to mark these tokens as used and prevents the customer from using them. The precise functioning of this protocol for refunding tokens remains to be determined in future work.

5.4. Verifiable credentials

Another subject worthy of future investigation is the evaluation of new signature schemes for tokens. Of particular interest are verifiable credential signature schemes, which permit selective disclosure of signed information. Such a signature scheme could enhance the degree of anonymity, as anonymity sets are no longer defined by a combination of token family and subscription period, but solely by token family. The customer could effectively prove that they possess a token that is still valid, without disclosing its expiration date. This would allow precise token expiration, without the necessity for token start date rounding. The principal disadvantage of verifiable credential signature schemes is their performance. Token issuance and usage are operations that can occur in rapid succession, therefore the performance of the signature scheme is critical. Furthermore, a not insignificant part of the computation happens on the device of the customer, which can include older mobile devices.

Since the publication of the Camenisch-Lysyanskaya (CL) signature scheme in [26], there have been new verifiable credential schemes that offer enhanced performance and shorter keys. One such promising example is the BBS+ signature scheme described in [27].

5.5. Termination of subscriptions

Due to their unlinkability, subscriptions cannot be terminated before their planned expiration date in a fine-grained manner. With the proposed solution, subscriptions are valid until they expire, even if they are stolen or lost. Therefore, merchants are recommended to limit the duration of subscriptions (e.g., to one month) to limit the damage done if a subscription is lost or stolen. In the event that a merchant's token issue key is stolen or made publicly available, the merchant may choose to cease accepting all tokens signed by that key.

5.6. Practical deployment

In scenarios where rotating a token for every single subscription usage is infeasible or impractical, merchants could deploy the proposed solution to allow for a compromise between privacy and practicality. This would entail customers using their subscription token to obtain a short-lived session cookie, which would allow them to access the subscription goods during the validity of the cookie. This would facilitate the utilization of the subscription during the period of validity of the aforementioned cookie. If the lifetime of the session cookie is sufficiently brief (between 30 minutes and one hour), the merchant's ability to profile the customer is still constrained, and sharing is similarly impractical.

6. Conclusion

This thesis introduces a practical solution for privacy-preserving, token-based subscriptions and discounts.

The internet, as it stands today, is heavily reliant on advertising, which drives consumption. Independent journalists often depend on people buying products through their referral links, and news outlets focus on driving user engagement, sometimes at the expense of journalistic quality. This dependence on consumerism compromises the integrity of journalism.

If independent journalists and news outlets could receive direct payments from readers through a privacy-preserving channel, the distortion caused by advertising would diminish. This would pave the way for better, independent, and critical coverage of pressing topics. While payment systems like GNU Taler already enable microtransactions for individual articles, the solution presented in this thesis supports sustainable subscription-based business models, providing a direct and consistent revenue stream.

In essence, the work presented in this thesis can be considered a small piece in a much larger puzzle aimed at reshaping the digital economy. It can help foster more direct and equitable financial interactions between consumers and content creators, and pave the way for a myriad of new applications in our increasingly tokenized world. Tokens can represent more than just money, subscriptions or discounts. They have the potential to be used for asset management (stocks, index funds, or securities) or as donation receipts [28]. Multi-input-multi-output contracts can be used to distribute dividends across stakeholders, and tokens could be used to distribute voting rights for decisions at a company.

Declaration of Authorship

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those acknowledged.

All statements taken from other writings, either literally or in essence, have been marked as such.

I hereby agree that the present work may be reviewed in electronic form using appropriate software.

[illegible]

June 13, 2024 C. Blaettler

Bibliography

[1] Charles Duhigg. How companies learn your secrets. *The New York Times Magazine*.

[2] Ellie House. Netflix: How did it know i was bi before i did? *BBC*.

[3] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.

[4] Jerry Lambe. New york times makes subscription cancellations exceedingly difficult for consumers, class action lawsuit alleges. *Law & Crime*.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. *Paul Syverson*, 13, 06 2004.

[6] Marina Blanton. Online subscriptions with anonymous access. pages 217–227, 03 2008.

[7] Andreas Pfitzmann and Marit Köhntopp. *Anonymity, Unobservability, and Pseudonymity — A Proposal for Terminology*, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[8] Volker Fusenig, Eugen Staab, Uli Sorger, and Thomas Engel. Unlinkable communication. In *2008 Sixth Annual Conference on Privacy, Security and Trust*, pages 51–55. IEEE, 2008.

[9] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, pages 41–53, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[10] Claudia Díaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, pages 54–68, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[11] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed elgamal encryption in the algebraic group model. Cryptology ePrint Archive, Paper 2019/877, 2019. `https://eprint.iacr.org/2019/877`.

[12] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clone wars:

efficient periodic n-times anonymous authentication. Cryptology ePrint Archive, Paper 2006/454, 2006. `https://eprint.iacr.org/2006/454`.

[13] Stuart G Stubblebine, Paul F Syverson, and David M Goldschlag. Unlinkable serial transactions: protocols and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2(4):354–389, 1999.

[14] Zulfikar Ramzan and Matthias Ruhl. Protocols for anonymous subscription services. *Unpublished manuscript*, 39, 2000.

[15] Toru Nakanishi, Nobuaki Haruna, and Yuji Sugiyama. Unlinkable electronic coupon protocol with anonymity control. In *Information Security*, pages 37–46, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[16] Liqun Chen, Matthias Enzmann, Ahmad-Reza Sadeghi, Markus Schneider, and Michael Steiner. A privacy-protecting coupon system. In Andrew S. Patrick and Moti Yung, editors, *Financial Cryptography and Data Security*, pages 93–108, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] Weiwei Liu, Yi Mu, Guomin Yang, and Yong Yu. Efficient e-coupon systems with strong user privacy. *Telecommunication Systems*, 64:695–708, 2017.

[18] Liqun Chen, Alberto N Escalante B, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. A privacy-protecting multi-coupon scheme with stronger protection against splitting. In *Financial Cryptography and Data Security: 11th International Conference, FC 2007, and 1st International Workshop on Usable Security, USEC 2007, Scarborough, Trinidad and Tobago, February 12-16, 2007. Revised Selected Papers 11*, pages 29–44. Springer, 2007.

[19] Quynh Dang. Secure hash standard, 2015-08-04 2015.

[20] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 319–327, New York, NY, 1990. Springer New York.

[21] Daniel J. Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. Eddsa for more curves. Cryptology ePrint Archive, Paper 2015/677, 2015. `https://eprint.iacr.org/2015/677`.

[22] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Paper 2011/368, 2011. `https://eprint.iacr.org/2011/368`.

[23] A. Philips and M. Davis. Tags for identifying languages. BCP 47, RFC Editor, september 2009.

[24] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. Uniform resource identifier (uri): Generic syntax. Technical Report 3986, January 2005.

[25] Mark Nottingham. Http cache-control extensions for stale content. Technical Report 5861, RFC Editor, 5 2010.

[26] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, pages 268–289, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[27] Stefano Tessaro and Chenzhi Zhu. Revisiting bbs signatures. Cryptology ePrint Archive, Paper 2023/275, 2023. `https://eprint.iacr.org/2023/275`.

[28] Johannes Casaburi and Lukas Matyja. Donau: Donation authority. tax-deductable privacy-preserving donations.

A. Merchant REST API documentation

This is an excerpt from the merchant backend API documentation.[1] It was written by the GNU Taler community. During the course of this thesis, I made additions and modifications to the existing documentation. All endpoints relevant to this thesis are included here for the sake of completeness.

A.1. Wallet API

This section describes (public) endpoints that wallets must be able to interact with directly (without HTTP-based authentication). These endpoints are used to process payments (claiming an order, paying for the order, checking payment/refund status and aborting payments), and to process refunds (checking refund status, obtaining the refund).

Claiming an order

The first step of processing any Taler payment consists of the (authorized) wallet claiming the order for itself. In this process, the wallet provides a wallet-generated nonce that is added into the contract terms. This step prevents two different wallets from paying for the same contract, which would be bad especially if the merchant only has finite stocks.

A claim token can be used to ensure that the wallet claiming an order is actually authorized to do so. This is useful in cases where order IDs are predictable and malicious actors may try to claim orders (say in a case where stocks are limited).

POST `[/instances/$INSTANCE]/orders/$ORDER_ID/claim`

> Wallet claims ownership (via nonce) over an order. By claiming an order, the wallet obtains the full contract terms, and thereby implicitly also the hash of the contract terms it needs for the other `public` APIs to authenticate itself as the wallet that is indeed eligible to inspect this particular orders status.
>
> **Request:**
>
> The request must be a `ClaimRequest`.

[1]`https://docs.taler.net/core/api-merchant.html`

```
interface ClaimRequest {
  // Nonce to identify the wallet that claimed the order.
  nonce: string;

  // Token that authorizes the wallet to claim the order.
  // *Optional* as the merchant may not have required it
  // (create_token set to false in PostOrderRequest).
  token?: ClaimToken;
}
```

Response:

200 OK: The client has successfully claimed the order. The response contains the contract terms.

404 Not Found: The backend is unaware of the instance or order.

409 Conflict: Someone else has already claimed the same order ID with a different nonce.

```
interface ClaimResponse {
  // Contract terms of the claimed order
  contract_terms: ContractTerms;

  // Signature by the merchant over the contract terms.
  sig: EddsaSignature;
}
```

Making the payment

POST `[/instances/$INSTANCE]/orders/$ORDER_ID/pay` Pay for an order by giving a deposit permission for coins. Typically used by the customer's wallet. Note that this request does not include the usual `h_contract` argument to authenticate the wallet, as the hash of the contract is implied by the signatures of the coins. Furthermore, this API doesn't really return useful information about the order.

Request:

The request must be a pay request.

Response:

200 OK: The exchange accepted all of the coins. The body is a payment response. The `frontend` should now fulfill the contract. Note that it is possible that refunds have been granted.

400 Bad request: Either the client request is malformed or some specific processing error happened that may be the fault of the client as detailed in the JSON body of the response. This includes the case where the payment is insufficient (sum is below the required total amount, for example because the wallet calculated the fees wrong).

402 Payment required: There used to be a sufficient payment, but due to refunds the amount effectively paid is no longer sufficient. (If the amount is generally insufficient, we return "400 Bad Request", only if this is because of refunds we return 402.)

403 Forbidden: One of the coin signatures was not valid.

404 Not found: The merchant backend could not find the order or the instance and thus cannot process the payment.

408 Request timeout: The backend took too long to process the request. Likely the merchant's connection to the exchange timed out. Try again.

409 Conflict: The exchange rejected the payment because a coin was already spent (or used in a different way for the same purchase previously), or the merchant rejected the payment because the order was already fully paid (and then return signatures with refunds). If a coin was already spent (this includes re-using the same coin after a refund), the response will include the `exchange_url` for which the payment failed, in addition to the response from the exchange to the `/batch-deposit` request.

410 Gone: The offer has expired and is no longer available.

412 Precondition failed: The given exchange is not acceptable for this merchant, as it is not in the list of accepted exchanges and not audited by an approved auditor. TODO: Status code may be changed to 409 in the future as 412 is technically wrong.

502 Bad gateway: The merchant's interaction with the exchange failed in some way. The client might want to try again later. This includes failures such as the denomination key of a coin not being known to the exchange as far as the merchant can tell.

504 Gateway timeout: The merchant's interaction with the exchange took too long. The client might want to try again later.

The backend will return verbatim the error codes received from the exchange's deposit API. If the wallet made a mistake, like by double-spending for example, the frontend should pass the reply verbatim to the browser/wallet. If the payment was successful, the frontend MAY use this to trigger some business logic.

```
interface PaymentResponse {
    // Signature on TALER_PaymentResponsePS with the public
    // key of the merchant instance.
    sig: EddsaSignature;

    // Text to be shown to the point-of-sale staff as a proof
    ↪  of
    // payment.
    pos_confirmation?: string;

    // Signed tokens. Returned in the same order as the
    // token envelopes were provided in the request.
    // @since protocol **vSUBSCRIBE**
    token_sigs?: SignedTokenEnvelope[];

  }

interface PayRequest {
    // The coins used to make the payment.
    coins: CoinPaySig[];

    // Input tokens required by choice indicated by
    ↪  choice_index.
    // @since protocol **vSUBSCRIBE**
    tokens?: TokenUseSig[];

    // Array of output tokens to be (blindly) signed by the
    ↪  merchant.
    // Output tokens specified in choice indicated by
    ↪  choice_index.
    // @since protocol **vSUBSCRIBE**
    tokens_evs?: TokenEnvelope[];

    // Custom inputs from the wallet for the contract.
    wallet_data?: PayWalletData;

    // The session for which the payment is made (or
    ↪  replayed).
    // Only set for session-based payments.
    session_id?: string;

  }

interface SignedTokenEnvelope {

    // Blind signature made by the merchant.
    blind_sig: TokenIssueBlindSig;

  }

type TokenIssueBlindSig = RSATokenIssueBlindSig |
↪  CSTokenIssueBlindSig;
```

```
interface RSATokenIssueBlindSig {
    cipher: "RSA";

    // (blinded) RSA signature
    blinded_rsa_signature: BlindedRsaSignature;
  }


interface CSTokenIssueBlindSig {
    cipher: "CS";

    // Signer chosen bit value, 0 or 1, used
    // in Clause Blind Schnorr to make the
    // ROS problem harder.
    b: Integer;

    // Blinded scalar calculated from c_b.
    s: Cs25519Scalar;

  }


interface PayWalletData {
    // Index of the selected choice within the choices array
     ↪  of
    // the contract terms.
    // @since protocol **vSUBSCRIBE**
    choice_index?: Integer;

    // Output commitment. Hash over output token envelopes.
    // @since protocol **vSUBSCRIBE**
    h_outputs?: HashCode;
  }


interface CoinPaySig {
    // Signature by the coin.
    coin_sig: EddsaSignature;

    // Public key of the coin being spent.
    coin_pub: EddsaPublicKey;

    // Signature made by the denomination public key.
    ub_sig: UnblindedSignature;

    // The hash of the denomination public key associated with
     ↪  this coin.
    h_denom: HashCode;
    // The amount that is subtracted from this coin with this
    ↪  payment.
    contribution: Amount;

    // URL of the exchange this coin was withdrawn from.
    exchange_url: string;

    // Signature affirming the posession of the
    // respective private key proving that the payer
    // is old enough. Only provided if the paid contract
    // has an age restriction and the coin is
    // age-restricted.
    minimum_age_sig?: EddsaSignature;

    // Age commitment vector of the coin.
    // Only provided if the paid contract
    // has an age restriction and the coin is
    // age-restricted.
    age_commitment?: Edx25519PublicKey[];

    // Hash over the agge commitment vector of the coin.
    // Only provided if the paid contract
    // does NOT have an age restriction and the coin is
    // age-restricted.
    h_age_commitment?: AgeCommitmentHash;
  }


interface TokenUseSig {

    // Signature on TALER_TokenUseRequestPS with the token use
    ↪  key of
    // the token being used in this request.
    token_sig: EddsaSignature;

    // Token use public key.
    token_pub: EddsaPublicKey;

    // Unblinded signature on TALER_TokenIssueRequestPS with
    ↪  the token
    // issue public key of the merchant.
    ub_sig: UnblindedSignature;

  }
```

```
// This type depends on the cipher used to sign token
↪ families. This is
  // configured by the merchant and defined for each token
  ↪ family in the
  // contract terms.
  type TokenEnvelope = RSATokenEnvelope | CSTokenEnvelope;


interface RSATokenEnvelope {

    // RSA is used for the blind signature.
    cipher: "RSA";

    // Blinded signature of the token's public EdDSA key.
    rsa_blinded_pub: BlindedRsaSignature;

  }


interface CSTokenEnvelope {
    // Blind Clause-Schnorr signature scheme is used for the
    ↪ blind signature.
    // See https://taler.net/papers/cs-thesis.pdf for details.
    cipher: "CS";

    // Public nonce
    cs_nonce: string;       // Crockford Base32 encoded

    // Two Curve25519 scalars, each representing a blinded
    ↪ challenge
    cs_blinded_c0: string; // Crockford Base32 encoded
    cs_blinded_c1: string; // Crockford Base32 encoded
  }
```

Querying payment status

GET[/instances/$INSTANCE]/orders/$ORDER_ID Query the payment status of an order. This endpoint is for the wallet. When the wallet goes to this URL and it is unpaid, it will be prompted for payment. This endpoint typically also supports requests with the “Accept” header requesting “text/html”. In this case, an HTML response suitable for triggering the interaction with the wallet is returned, with `timeout_ms` ignored (treated as zero). If the backend installation does not include the required HTML templates, a 406 status code is returned.

In the case that the request was made with a claim token (even the wrong one) and the order was claimed and paid, the server will redirect the client to the fulfillment URL. This redirection will happen with a 302 status code if the “Accept” header specified “text/html”, and with a 202 status code otherwise.

Request:

Query Parameters:

- **h_contract=HASH** – *Optional.* Hash of the order’s contract terms (this is used to authenticate the wallet/customer in case $ORDER_ID is guessable). Required once an order was claimed.
- **token=TOKEN** – *Optional.* Authorizes the request via the claim token that was returned in the PostOrderResponse. Used with unclaimed orders only. Whether token authorization is required is determined by the merchant when the frontend creates the order.
- **session_id=STRING** – *Optional.* Session ID that the payment must be bound to. If not specified, the payment is not session-bound.
- **timeout_ms=NUMBER** – *Optional.* If specified, the merchant backend will wait up to `timeout_ms` milliseconds for completion of the payment before sending the HTTP response. A client must never rely on this behavior, as the merchant backend may return a response immediately.
- **await_refund_obtained=BOOLEAN** – *Optional.* If set to “yes”, poll for the order’s pending refunds to be picked up. `timeout_ms` specifies how long we will wait for the refund.
- **refund=AMOUNT** – *Optional.* Indicates that we are polling for a refund above the given AMOUNT. `timeout_ms` will specify how long we will wait for the refund.
- **allow_refunded_for_repurchase** – *Optional.* Since protocol **v9** refunded orders are only returned under “already_paid_order_id” if this flag is set explicitly to “YES”.

Response:

200 OK: The response is a StatusPaidResponse.

202 Accepted: The response is a StatusGotoResponse. Only returned if the content type requested was not HTML. The target site may allow the client to setup a fresh order (as this one has been taken) or may trigger repurchase detection.

302 Found: The client should go to the indicated location (via HTTP “Location:” header). Only returned if the content type requested was HTML. The target site may allow the client to setup a fresh order (as this one has been taken) or may trigger repurchase detection.

402 Payment required: The response is a StatusUnpaidResponse.

403 Forbidden: The `h_contract` (or the `token` for unclaimed orders) does not match the order and we have no fulfillment URL in the contract.

404 Not found: The merchant backend is unaware of the order.

406 Not acceptable: The merchant backend could not load the template required to generate a reply in the desired format. (Likely HTML templates were not properly installed.)

```
interface StatusPaidResponse {
    // Was the payment refunded (even partially, via refund or
    ↪  abort)?
    refunded: boolean;

    // Is any amount of the refund still waiting to be picked
    ↪  up (even partially)?
    refund_pending: boolean;

    // Amount that was refunded in total.
    refund_amount: Amount;

    // Amount that already taken by the wallet.
    refund_taken: Amount;
  }


interface StatusGotoResponse {
    // The client should go to the reorder URL, there a fresh
    // order might be created as this one is taken by another
    // customer or wallet (or repurchase detection logic may
    // apply).
    public_reorder_url: string;
  }


interface StatusUnpaidResponse {
    // URI that the wallet must process to complete the
    ↪  payment.
    taler_pay_uri: string;

    // Status URL, can be used as a redirect target for the
    ↪  browser
    // to show the order QR code / trigger the wallet.
    fulfillment_url?: string;

    // Alternative order ID which was paid for already in the
    ↪  same session.
    // Only given if the same product was purchased before in
    ↪  the same session.
    already_paid_order_id?: string;
  }
```

A.2. Payment processing

To process Taler payments, a merchant must first set up an order with the merchant backend. The order is then claimed by a wallet, and paid by the wallet. The merchant can check the payment status of the order. Once the order is paid, the merchant may (for a limited time) grant refunds on the order.

Creating orders

POST [/instances/$INSTANCE]/private/orders Create a new order that a customer can pay for.

This request is **not** idempotent unless an `order_id` is explicitly specified. However, while repeating without an `order_id` will create another order, that is generally pretty harmless (as long as only one of the orders is returned to the wallet).

Note: This endpoint does not return a URL to redirect your user to confirm the payment. In order to get this URL use `GET [/instances/$INSTANCE]/orders/$ORDER_ID`. The API is structured this way since the payment redirect URL is not unique for every order, there might be varying parameters such as the session id.

Request:

The request must be a PostOrderRequest.

Response:

200 OK: The backend has successfully created the proposal. The response is a PostOrderResponse.

404 Not found: Possible reasons are:

1. The order given used products from the inventory, but those were not found in the inventory.
2. The merchant instance is unknown (including possibly the instance being not configured for new orders).
3. The wire method specified is not supported by the backend.
4. An OTP device ID was specified and is unknown.

Details in the error code. NOTE: currently the client has no good way to find out which product is not in the inventory, we MAY want to specify that in the reply.

409 Conflict: A different proposal already exists under the specified order ID, or the requested currency is not supported by this backend. Details in the error code.

410 Gone: The order given used products from the inventory that are out of stock. The response is a OutOfStockResponse.

```
interface PostOrderRequest {
    // The order must at least contain the minimal
    // order detail, but can override all.
    order: Order;

    // If set, the backend will then set the refund deadline to
     ↪  the current
    // time plus the specified delay.  If it's not set, refunds
     ↪  will not be
    // possible.
    refund_delay?: RelativeTime;

    // Specifies the payment target preferred by the client.
     ↪  Can be used
    // to select among the various (active) wire methods
     ↪  supported by the instance.
    payment_target?: string;

    // The session for which the payment is made (or
     ↪  replayed).
    // Only set for session-based payments.
    // Since protocol **v6**.
    session_id?: string;

    // Specifies that some products are to be included in the
    // order from the inventory.  For these inventory
     ↪  management
    // is performed (so the products must be in stock) and
    // details are completed from the product data of the
     ↪  backend.
    inventory_products?: MinimalInventoryProduct[];

    // Specifies a lock identifier that was used to
    // lock a product in the inventory.  Only useful if
    // inventory_products is set.  Used in case a frontend
    // reserved quantities of the individual products while
    // the shopping cart was being built.  Multiple UUIDs can
    // be used in case different UUIDs were used for different
    // products (i.e. in case the user started with multiple
    // shopping sessions that were combined during checkout).
    lock_uuids?: string[];

    // Should a token for claiming the order be generated?
    // False can make sense if the ORDER_ID is sufficiently
    // high entropy to prevent adversarial claims (like it is
    // if the backend auto-generates one). Default is 'true'.
    // Note: This is NOT related to tokens used for
    ↪  subscriptins or discounts.
    create_token?: boolean;

    // OTP device ID to associate with the order.
    // This parameter is optional.
    otp_id?: string;
  }
```

The Order object represents the starting point for new ContractTerms. After validating and sanatizing all inputs, the merchant backend will add additional information to the order and create a new ContractTerms object that will be stored in the database.

```
type Order = (OrderV1 | OrderV0) & OrderCommon;


interface OrderV1 {
    // Version 1 order support discounts and subscriptions.
    //
    ↪  https://docs.taler.net/design-documents/046-mumimo-contracts.html
    // @since protocol **vSUBSCRIBE**
    version: 1;

    // List of contract choices that the customer can select
    ↪  from.
    // @since protocol **vSUBSCRIBE**
    choices?: OrderChoice[];
  }


interface OrderV0 {
    // Optional, defaults to 0 if not set.
    version?: 0;

    // Total price for the transaction. The exchange will
    ↪  subtract deposit
    // fees from that amount before transferring it to the
    ↪  merchant.
    amount: Amount;
    // Maximum total deposit fee accepted by the merchant for
    ↪  this contract.
    // Overrides defaults of the merchant instance.
    max_fee?: Amount;
  }


interface OrderCommon {
    // Human-readable description of the whole purchase.
    summary: string;

    // Map from IETF BCP 47 language tags to localized
    ↪  summaries.
    summary_i18n?: { [lang_tag: string]: string };

    // Unique, free-form identifier for the order.
    // Must be unique within a merchant instance.
    // For merchants that do not store proposals in their DB
    // before the customer paid for them, the order_id can be
    ↪  used
    // by the frontend to restore a proposal from the
    ↪  information
    // encoded in it (such as a short product identifier and
    ↪  timestamp).
    order_id?: string;

    // URL where the same contract could be ordered again (if
    // available). Returned also at the public order endpoint
    // for people other than the actual buyer (hence public,
    // in case order IDs are guessable).
    public_reorder_url?: string;

    // See documentation of fulfillment_url field in
    ↪  ContractTerms.
    // Either fulfillment_url or fulfillment_message must be
    ↪  specified.
    // When creating an order, the fulfillment URL can
    // contain ${ORDER_ID} which will be substituted with the
    // order ID of the newly created order.
    fulfillment_url?: string;

    // See documentation of fulfillment_message in
    ↪  ContractTerms.
    // Either fulfillment_url or fulfillment_message must be
    ↪  specified.
    fulfillment_message?: string;
```

```
// Map from IETF BCP 47 language tags to localized
↪  fulfillment
// messages.
fulfillment_message_i18n?: { [lang_tag: string]: string };

// List of products that are part of the purchase.
products?: Product[];

// Time when this contract was generated. If null, defaults
↪  to current
// time of merchant backend.
timestamp?: Timestamp;

// After this deadline has passed, no refunds will be
↪  accepted.
// Overrides deadline calculated from refund_delay in
// PostOrderRequest.
refund_deadline?: Timestamp;

// After this deadline, the merchant won't accept payments
↪  for the contract.
// Overrides deadline calculated from default pay delay
↪  configured in
// merchant backend.
pay_deadline?: Timestamp;

// Transfer deadline for the exchange. Must be in the
↪  deposit permissions
// of coins used to pay for this order.
// Overrides deadline calculated from default wire transfer
↪  delay
// configured in merchant backend. Must be after refund
↪  deadline.
wire_transfer_deadline?: Timestamp;

// Base URL of the (public!) merchant backend API.
// Must be an absolute URL that ends with a slash.
// Defaults to the base URL this request was made to.
merchant_base_url?: string;

// Delivery location for (all!) products.
delivery_location?: Location;

// Time indicating when the order should be delivered.
// May be overwritten by individual products.
// Must be in the future.
```

```
    delivery_date?: Timestamp;

    // See documentation of auto_refund in ContractTerms.
    // Specifies for how long the wallet should try to get an
    // automatic refund for the purchase.
    auto_refund?: RelativeTime;

    // Extra data that is only interpreted by the merchant
     ↪  frontend.
    // Useful when the merchant needs to store extra
     ↪  information on a
    // contract without storing it separately in their
     ↪  database.
    // Must really be an Object (not a string, integer, float
     ↪  or array).
    extra?: Object;
  }
```

The OrderChoice object describes a possible choice within an order. The choice is done by the wallet and consists of in- and outputs. In the example of buying an article, the merchant could present the customer with the choice to use a valid subscription token or pay using a gift voucher. Available since protocol **vSUBSCRIBE**.

```
interface OrderChoice {
    // Total price for the choice. The exchange will subtract
     ↪  deposit
    // fees from that amount before transferring it to the
     ↪  merchant.
    amount: Amount;

    // Inputs that must be provided by the customer, if this
     ↪  choice is selected.
    // Defaults to empty array if not specified.
    inputs?: OrderInput[];

    // Outputs provided by the merchant, if this choice is
     ↪  selected.
    // Defaults to empty array if not specified.
    outputs?: OrderOutput[];

    // Maximum total deposit fee accepted by the merchant for
     ↪  this contract.
    // Overrides defaults of the merchant instance.
    max_fee?: Amount;
  }
```

```
// For now, only token inputs are supported.
  type OrderInput = OrderInputToken;

interface OrderInputToken {

    // Token input.
    type: "token";

    // Token family slug as configured in the merchant backend.
    ↪  Slug is unique
    // across all configured tokens of a merchant.
    token_family_slug: string;

    // How many units of the input are required.
    // Defaults to 1 if not specified. Output with count == 0
    ↪  are ignored by
    // the merchant backend.
    count?: Integer;

  }

type OrderOutput = OrderOutputToken | OrderOutputTaxReceipt;

interface OrderOutputToken {

    // Token output.
    type: "token";

    // Token family slug as configured in the merchant backend.
    ↪  Slug is unique
    // across all configured tokens of a merchant.
    token_family_slug: string;

    // Start of the validity period of the token. Based on
    ↪  this, the merchant
    // will select the relevant signing key. This value is
    ↪  rounded down
    // according to the configured rounding duration in the
    ↪  token family.
    // If not specified, the current time is used.
    valid_after?: Timestamp;
```

```
      // How many units of the output are issued by the
      ↪  merchant.
      // Defaults to 1 if not specified. Output with count == 0
      ↪  are ignored by
      // the merchant backend.
      count?: Integer;

    }
```

```
  interface OrderOutputTaxReceipt {

      // Tax receipt output.
      type: "tax-receipt";

      // Base URL of the donation authority that will
      // issue the tax receipt.
      donau_url: string;

    }
```

The following MinimalInventoryProduct can be provided if the parts of the order are inventory-based, that is if the PostOrderRequest uses `inventory_products`. For such products, which must be in the backend's inventory, the backend can automatically fill in the amount and other details about the product that are known to it from its `products` table. Note that the `inventory_products` will be appended to the list of `products` that the frontend already put into the `order`. So if the frontend can sell additional non-inventory products together with `inventory_products`. Note that the backend will NOT update the `amount` of the `order`, so the frontend must already have calculated the total price — including the `inventory_products`.

```
// Note that if the frontend does give details beyond these,
  // it will override those details (including price or taxes)
  // that the backend would otherwise fill in via the inventory.
  interface MinimalInventoryProduct {
    // Which product is requested (here mandatory!).
    product_id: string;

    // How many units of the product are requested.
    quantity: Integer;
  }


interface PostOrderResponse {
    // Order ID of the response that was just created.
    order_id: string;

    // Token that authorizes the wallet to claim the order.
    // Provided only if "create_token" was set to 'true'
    // in the request.
    token?: ClaimToken;
  }
```

```
interface OutOfStockResponse {

    // Product ID of an out-of-stock item.
    product_id: string;

    // Requested quantity.
    requested_quantity: Integer;

    // Available quantity (must be below requested_quantity).
    available_quantity: Integer;

    // When do we expect the product to be again in stock?
    // Optional, not given if unknown.
    restock_expected?: Timestamp;
  }
```

Inspecting orders

GET [/instances/$INSTANCE]/private/orders Returns known orders up to some point in the past.

Request:

Query Parameters:

- **paid** – *Optional.* If set to yes, only return paid orders, if no only unpaid orders. Do not give (or use “all”) to see all orders regardless of payment status.
- **refunded** – *Optional.* If set to yes, only return refunded orders, if no only unrefunded orders. Do not give (or use “all”) to see all orders regardless of refund status.
- **wired** – *Optional.* If set to yes, only return wired orders, if no only orders with missing wire transfers. Do not give (or use “all”) to see all orders regardless of wire transfer status.
- **delta** – *Optional.* takes value of the form `N (-N)`, so that at most `N` values strictly older (younger) than `start` and `date_s` are returned.

Defaults to `-20` to return the last 20 entries (before `start` and/or `date_s`). Deprecated in protocol **v12**. Use *limit* instead.

- **limit** – *Optional.* At most return the given number of results. Negative for descending by row ID, positive for ascending by row ID. Default is `20`. Since protocol **v12**.
- **date_s** – *Optional.* Non-negative date in seconds after the UNIX Epoc, see `delta` for its interpretation. If not specified, we default to the oldest or most recent entry, depending on `delta`.
- **start** – *Optional.* Row number threshold, see `delta` for its interpretation. Defaults to `INT64_MAX`, namely the biggest row id possible in the database. Deprecated in protocol **v12**. Use *offset* instead.
- **offset** – *Optional.* Starting `row_id` for an iteration. Since protocol **v12**.
- **timeout_ms** – *Optional.* Timeout in milliseconds to wait for additional orders if the answer would otherwise be negative (long polling). Only useful if delta is positive. Note that the merchant MAY still return a response that contains fewer than `delta` orders.
- **session_id** – *Optional.* Since protocol **v6**. Filters by session ID.
- **fulfillment_url** – *Optional.* Since protocol **v6**. Filters by fulfillment URL.

Response:

200 OK: The response is an OrderHistory.

```
interface OrderHistory {
    // Timestamp-sorted array of all orders matching the
    ↪  query.
    // The order of the sorting depends on the sign of delta.
    orders : OrderHistoryEntry[];
  }


interface OrderHistoryEntry {

    // Order ID of the transaction related to this entry.
    order_id: string;

    // Row ID of the order in the database.
    row_id: number;

    // When the order was created.
    timestamp: Timestamp;

    // The amount of money the order is for.
  amount: Amount;

  // The summary of the order.
  summary: string;

  // Whether some part of the order is refundable,
  // that is the refund deadline has not yet expired
  // and the total amount refunded so far is below
  // the value of the original transaction.
  refundable: boolean;

  // Whether the order has been paid or not.
  paid: boolean;
}
```

GET [/instances/$INSTANCE]/private/orders/$ORDER_ID Merchant checks the payment status of an order. If the order exists but is not paid and not claimed yet, the response provides a redirect URL. When the user goes to this URL, they will be prompted for payment. Differs from the `public` API both in terms of what information is returned and in that the wallet must provide the contract hash to authenticate, while for this API we assume that the merchant is authenticated (as the endpoint is not `public`).

Request:

Query Parameters:

- **session_id** – *Optional.* Session ID that the payment must be bound to. If not specified, the payment is not session-bound.
- **transfer** – Deprecated in protocol **V6**. *Optional.* If set to "YES", try to obtain the wire transfer status for this order from the exchange. Otherwise, the wire transfer status MAY be returned if it is available.
- **timeout_ms** – *Optional.* Timeout in milliseconds to wait for a payment if the answer would otherwise be negative (long polling).
- **allow_refunded_for_repurchase** – *Optional.* Since protocol **v9** refunded orders are only returned under "already_paid_order_id" if this flag is set explicitly to "YES".

Response:

200 OK: Returns a MerchantOrderStatusResponse, whose format can differ based on the status of the payment.

404 Not found: The order or instance is unknown to the backend.

502 Bad gateway: We failed to obtain a response from the exchange (about the wire transfer status).

504 Gateway timeout: The merchant's interaction with the exchange took too long. The client might want to try again later.

```
type MerchantOrderStatusResponse = CheckPaymentPaidResponse |
                                     CheckPaymentClaimedResponse
                                     ↪   |
                                     CheckPaymentUnpaidResponse;


interface CheckPaymentPaidResponse {
    // The customer paid for this contract.
    order_status: "paid";

    // Was the payment refunded (even partially)?
    refunded: boolean;

    // True if there are any approved refunds that the wallet
    ↪  has
    // not yet obtained.
    refund_pending: boolean;

    // Did the exchange wire us the funds?
    wired: boolean;

    // Total amount the exchange deposited into our bank
    ↪  account
    // for this contract, excluding fees.
    deposit_total: Amount;

    // Numeric error code indicating errors the exchange
    // encountered tracking the wire transfer for this purchase
    ↪  (before
    // we even got to specific coin issues).
    // 0 if there were no issues.
    exchange_code: number;

    // HTTP status code returned by the exchange when we asked
    ↪  for
    // information to track the wire transfer for this
    ↪  purchase.
    // 0 if there were no issues.
    exchange_http_status: number;

    // Total amount that was refunded, 0 if refunded is false.
    refund_amount: Amount;

    // Contract terms.
    contract_terms: ContractTerms;

    // Index of the selected choice within the choices array
     ↪  of
    // contract terms.
    // @since protocol **vSUBSCRIBE**
    choice_index?: Integer;

    // If the order is paid, set to the last time when a
     ↪  payment
    // was made to pay for this order. Since **v14**.
    last_payment: Timestamp;

    // The wire transfer status from the exchange for this
     ↪  order if
    // available, otherwise empty array.
    wire_details: TransactionWireTransfer[];

    // Reports about trouble obtaining wire transfer details,
    // empty array if no trouble were encountered.
    // @deprecated in protocol **v6**.
    wire_reports: TransactionWireReport[];

    // The refund details for this order.  One entry per
    // refunded coin; empty array if there are no refunds.
    refund_details: RefundDetails[];

    // Status URL, can be used as a redirect target for the
     ↪  browser
    // to show the order QR code / trigger the wallet.
    order_status_url: string;
  }


interface CheckPaymentClaimedResponse {
    // A wallet claimed the order, but did not yet pay for the
     ↪  contract.
    order_status: "claimed";

    // Contract terms.
    contract_terms: ContractTerms;

  }


interface CheckPaymentUnpaidResponse {
    // The order was neither claimed nor paid.
    order_status: "unpaid";

    // URI that the wallet must process to complete the
     ↪  payment.
    taler_pay_uri: string;

    // when was the order created
    creation_time: Timestamp;

    // Order summary text.
    summary: string;

    // Total amount of the order (to be paid by the customer).
    total_amount: Amount;

    // Alternative order ID which was paid for already in the
     ↪  same session.
    // Only given if the same product was purchased before in
     ↪  the same session.
    already_paid_order_id?: string;

    // Fulfillment URL of an already paid order. Only given if
     ↪  under this
    // session an already paid order with a fulfillment URL
     ↪  exists.
    already_paid_fulfillment_url?: string;

    // Status URL, can be used as a redirect target for the
     ↪  browser
    // to show the order QR code / trigger the wallet.
    order_status_url: string;

    // We do we NOT return the contract terms here because they
     ↪  may not
    // exist in case the wallet did not yet claim them.
  }

interface RefundDetails {
    // Reason given for the refund.
    reason: string;

    // Set to true if a refund is still available for the
     ↪  wallet for this payment.
    pending: boolean;

    // When was the refund approved.
    timestamp: Timestamp;

    // Total amount that was refunded (minus a refund fee).
    amount: Amount;
  }

interface TransactionWireTransfer {
    // Responsible exchange.
    exchange_url: string;

    // 32-byte wire transfer identifier.
    wtid: Base32;

    // Execution time of the wire transfer.
    execution_time: Timestamp;

    // Total amount that has been wire transferred
    // to the merchant.
    amount: Amount;

    // Was this transfer confirmed by the merchant via the
    // POST /transfers API, or is it merely claimed by the
    ↪  exchange?
    confirmed: boolean;
  }

interface TransactionWireReport {
    // Numerical error code.
    code: number;

    // Human-readable error description.
    hint: string;

    // Numerical error code from the exchange.
    exchange_code: number;

    // HTTP status code received from the exchange.
    exchange_http_status: number;

    // Public key of the coin for which we got the exchange
    ↪  error.
    coin_pub: CoinPublicKey;
  }
```

A.3. Token Families: Subscriptions, Discounts

This API provides functionalities for the issuance, management, and revocation of token families. Tokens facilitate the implementation of subscriptions and discounts, engaging solely the merchant and the user. Each token family encapsulates details pertaining to its respective tokens, guiding the merchant's backend on the appropriate processing and handling.

Creating token families

POST [/instances/$INSTANCES]/private/tokenfamilies This is used to create a token family.

Request:

The request must be a TokenFamilyCreateRequest.

Response:

204 No content: The token family was created successfully.

404 Not found: The merchant backend is unaware of the instance.

```
interface TokenFamilyCreateRequest {

    // Identifier for the token family consisting of
    ↪  unreserved characters
    // according to RFC 3986.
    slug: string;

    // Human-readable name for the token family.
    name: string;

    // Human-readable description for the token family.
    description: string;

    // Optional map from IETF BCP 47 language tags to
    ↪  localized descriptions.
    description_i18n?: { [lang_tag: string]: string };

    // Start time of the token family's validity period.
    // If not specified, merchant backend will use the
    ↪  current time.
    valid_after?: Timestamp;

    // End time of the token family's validity period.
    valid_before: Timestamp;

    // Validity duration of an issued token.
```

```
      duration: RelativeTime;

      // Rounding granularity of generated keys.
      rounding: RelativeTime;

      // Kind of the token family.
      kind: TokenFamilyKind;

    }
```

```
enum TokenFamilyKind {
      Discount = "discount",
      Subscription = "subscription",
    }
```

Updating token families

PATCH [/instances/$INSTANCES]/private/tokenfamilies/$SLUG This is used to update a token family.

Request:

The request must be a TokenFamilyUpdateRequest.

Response:

200 OK: The token family was successsful updated. Returns a TokenFamilyDetails.

404 Not found: The merchant backend is unaware of the token family or instance.

```
 interface TokenFamilyUpdateRequest {

      // Human-readable name for the token family.
      name: string;

      // Human-readable description for the token family.
      description: string;

      // Optional map from IETF BCP 47 language tags to
       ↪  localized descriptions.
      description_i18n: { [lang_tag: string]: string };

      // Start time of the token family's validity period.
      valid_after: Timestamp;

      // End time of the token family's validity period.
```

```
    valid_before: Timestamp;

    // Validity duration of an issued token.
    duration: RelativeTime;

  }
```

Inspecting token families

GET [/instances/$INSTANCES]/private/tokenfamilies This is used to list all configured token families for an instance.

Response:

200 OK: The merchant backend has successfully returned all token families. Returns a TokenFamiliesList.

404 Not found: The merchant backend is unaware of the instance.

```
// TODO: Add pagination

    interface TokenFamiliesList {

      // All configured token families of this instance.
      token_families: TokenFamilySummary[];

    }


interface TokenFamilySummary {
      // Identifier for the token family consisting of
       ↪  unreserved characters
      // according to RFC 3986.
      slug: string;

      // Human-readable name for the token family.
      name: string;

      // Start time of the token family's validity period.
      valid_after: Timestamp;

      // End time of the token family's validity period.
      valid_before: Timestamp;

      // Kind of the token family.
      kind: TokenFamilyKind;
   }
```

GET [/instances/$INSTANCES]/private/tokenfamilies/$SLUG This is used to get detailed information about a specific token family.

Response:

200 OK: The merchant backend has successfully returned the detailed information about a specific token family. Returns a TokenFamilyDetails.

404 Not found: The merchant backend is unaware of the token family or instance.

The TokenFamilyDetails object describes a configured token family.

```
interface TokenFamilyDetails {

      // Identifier for the token family consisting of
       ↪  unreserved characters
      // according to RFC 3986.
      slug: string;

      // Human-readable name for the token family.
      name: string;

      // Human-readable description for the token family.
      description: string;

      // Optional map from IETF BCP 47 language tags to
       ↪  localized descriptions.
      description_i18n?: { [lang_tag: string]: string };

      // Start time of the token family's validity period.
      valid_after: Timestamp;

      // End time of the token family's validity period.
      valid_before: Timestamp;

      // Validity duration of an issued token.
      duration: RelativeTime;

      // Rounding granularity of generated keys.
      rounding: RelativeTime;

      // Kind of the token family.
      kind: TokenFamilyKind;

      // How many tokens have been issued for this family.
      issued: Integer;

      // How many tokens have been used for this family.
```

```
            used: Integer;

          }
```

Deleting token families

DELETE [/instances/$INSTANCES]/private/tokenfamilies/$SLUG This is used to delete a token family. Issued tokens of this family will not be spendable anymore.

Response:

204 No content: The backend has successfully deleted the token family.

404 Not found: The merchant backend is unaware of the token family or instance.

A.4. The Contract Terms

This section describes the overall structure of the contract terms that are the foundation for Taler payments.

The contract terms must have the following structure:

```
type ContractTerms = (ContractTermsV1 | ContractTermsV0) &
↪ ContractTermsCommon;


interface ContractTermsV1 {
    // Version 1 supports the choices array, see
    //
    ↪ https://docs.taler.net/design-documents/046-mumimo-contracts.html.
    // @since protocol **vSUBSCRIBE**
    version: 1;

    // List of contract choices that the customer can select from.
    // @since protocol **vSUBSCRIBE**
    choices: ContractChoice[];

    // Map of storing metadata and issue keys of
    // token families referenced in this contract.
    // @since protocol **vSUBSCRIBE**
    token_families: { [token_family_slug: string]:
    ↪ ContractTokenFamily };
  }
```

```
interface ContractTermsV0 {
    // Defaults to version 0.
    version?: 0;

    // Total price for the transaction.
    // The exchange will subtract deposit fees from that amount
    // before transferring it to the merchant.
    amount: Amount;

    // Maximum total deposit fee accepted by the merchant for this
     ↪  contract.
    // Overrides defaults of the merchant instance.
    max_fee: Amount;
  }


interface ContractTermsCommon {
    // Human-readable description of the whole purchase.
    summary: string;

    // Map from IETF BCP 47 language tags to localized summaries.
    summary_i18n?: { [lang_tag: string]: string };

    // Unique, free-form identifier for the proposal.
    // Must be unique within a merchant instance.
    // For merchants that do not store proposals in their DB
    // before the customer paid for them, the order_id can be used
    // by the frontend to restore a proposal from the information
    // encoded in it (such as a short product identifier and
     ↪  timestamp).
    order_id: string;

    // URL where the same contract could be ordered again (if
    // available). Returned also at the public order endpoint
    // for people other than the actual buyer (hence public,
    // in case order IDs are guessable).
    public_reorder_url?: string;

    // URL that will show that the order was successful after
    // it has been paid for.  Optional, but either fulfillment_url
    // or fulfillment_message must be specified in every
    // contract terms.
    //
    // If a non-unique fulfillment URL is used, a customer can only
    // buy the order once and will be redirected to a previous
     ↪  purchase
```

```
// when trying to buy an order with the same fulfillment URL a
 ↪  second
// time. This is useful for digital goods that a customer only
 ↪  needs
// to buy once but should be able to repeatedly download.
//
// For orders where the customer is expected to be able to make
// repeated purchases (for equivalent goods), the fulfillment
 ↪  URL
// should be made unique for every order. The easiest way to do
// this is to include a unique order ID in the fulfillment URL.
//
// When POSTing to the merchant, the placeholder text
 ↪  "${ORDER_ID}"
// is be replaced with the actual order ID (useful if the
// order ID is generated server-side and needs to be
// in the URL). Note that this placeholder can only be used
 ↪  once.
// Front-ends may use other means to generate a unique
 ↪  fulfillment URL.
fulfillment_url?: string;

// Message shown to the customer after paying for the order.
// Either fulfillment_url or fulfillment_message must be
 ↪  specified.
fulfillment_message?: string;

// Map from IETF BCP 47 language tags to localized fulfillment
// messages.
fulfillment_message_i18n?: { [lang_tag: string]: string };

// List of products that are part of the purchase (see
 ↪  Product).
products: Product[];

// Time when this contract was generated.
timestamp: Timestamp;

// After this deadline has passed, no refunds will be accepted.
refund_deadline: Timestamp;

// After this deadline, the merchant won't accept payments for
 ↪  the contract.
pay_deadline: Timestamp;

// Transfer deadline for the exchange.  Must be in the
// deposit permissions of coins used to pay for this order.
wire_transfer_deadline: Timestamp;

// Merchant's public key used to sign this proposal; this
↪  information
// is typically added by the backend. Note that this can be an
↪  ephemeral key.
merchant_pub: EddsaPublicKey;

// Base URL of the (public!) merchant backend API.
// Must be an absolute URL that ends with a slash.
merchant_base_url: string;

// More info about the merchant, see below.
merchant: Merchant;

// The hash of the merchant instance's wire details.
h_wire: HashCode;

// Wire transfer method identifier for the wire method
↪  associated with h_wire.
// The wallet may only select exchanges via a matching auditor
↪  if the
// exchange also supports this wire method.
// The wire transfer fees must be added based on this wire
↪  transfer method.
wire_method: string;

// Exchanges that the merchant accepts even if it does not
↪  accept any auditors that audit them.
exchanges: Exchange[];

// Delivery location for (all!) products.
delivery_location?: Location;

// Time indicating when the order should be delivered.
// May be overwritten by individual products.
delivery_date?: Timestamp;

// Nonce generated by the wallet and echoed by the merchant
// in this field when the proposal is generated.
nonce: string;

// Specifies for how long the wallet should try to get an
// automatic refund for the purchase. If this field is
// present, the wallet should wait for a few seconds after
```

```
    // the purchase and then automatically attempt to obtain
    // a refund.  The wallet should probe until "delay"
    // after the payment was successful (i.e. via long polling
    // or via explicit requests with exponential back-off).
    //
    // In particular, if the wallet is offline
    // at that time, it MUST repeat the request until it gets
    // one response from the merchant after the delay has expired.
    // If the refund is granted, the wallet MUST automatically
    // recover the payment.  This is used in case a merchant
    // knows that it might be unable to satisfy the contract and
    // desires for the wallet to attempt to get the refund without
     ↪  any
    // customer interaction.  Note that it is NOT an error if the
    // merchant does not grant a refund.
    auto_refund?: RelativeTime;

    // Extra data that is only interpreted by the merchant
     ↪  frontend.
    // Useful when the merchant needs to store extra information on
     ↪  a
    // contract without storing it separately in their database.
    // Must really be an Object (not a string, integer, float or
     ↪  array).
    extra?: Object;

    // Minimum age the buyer must have (in years). Default is 0.
    // This value is at least as large as the maximum over all
    // mimimum age requirements of the products in this contract.
    // It might also be set independent of any product, due to
    // legal requirements.
    minimum_age?: Integer;

  }


interface ContractChoice {
    // Price to be paid for this choice. Could be 0.
    // The price is in addition to other instruments,
    // such as rations and tokens.
    // The exchange will subtract deposit fees from that amount
    // before transferring it to the merchant.
    amount: Amount;

    // List of inputs the wallet must provision (all of them) to
    // satisfy the conditions for the contract.
    inputs: ContractInput[];
    // List of outputs the merchant promises to yield (all of them)
    // once the contract is paid.
    outputs: ContractOutput[];

    // Maximum total deposit fee accepted by the merchant for this
     ↪  contract.
    max_fee: Amount;
  }

// For now, only tokens are supported as inputs.
  type ContractInput = ContractInputToken;

interface ContractInputToken {
    type: "token";

    // Slug of the token family in the
    // 'token_families' map on the order.
    token_family_slug: string;

    // Start of the validity period of the token. This is used to
     ↪  find the
    // matching public key within the token family.
    valid_after: Timestamp;

    // Number of tokens of this type required.
    // Defaults to one if the field is not provided.
    number?: Integer;
  };

// For now, only tokens are supported as outputs.
  type ContractOutput = ContractOutputToken;

interface ContractOutputToken {
    type: "token";

    // Slug of the token family in the
    // 'token_families' map on the top-level.
    token_family_slug: string;

    // Start of the validity period of the token. This is used to
     ↪  find the
    // matching public key within the token family.
    valid_after: Timestamp;
    // Number of tokens to be issued.
    // Defaults to one if the field is not provided.
    number?: Integer;
  }

interface ContractTokenFamily {
    // Human-readable name of the token family.
    name: string;

    // Human-readable description of the semantics of
    // this token family (for display).
    description: string;

    // Map from IETF BCP 47 language tags to localized
     ↪  descriptions.
    description_i18n?: { [lang_tag: string]: string };

    // Public keys used to validate tokens issued by this token
     ↪  family.
    keys: TokenIssuePublicKey[];

    // Kind-specific information of the token
    details: ContractTokenDetails;

    // Must a wallet understand this token type to
    // process contracts that use or issue it?
    critical: boolean;
  };

type TokenIssuePublicKey =
    | TokenIssueRsaPublicKey
    | TokenIssueCsPublicKey;

interface TokenIssueRsaPublicKey {
    cipher: "RSA";

    // RSA public key.
    rsa_pub: RsaPublicKey;

    // Start time of this key's validity period.
    valid_after: Timestamp;

    // End time of this key's validity period.
    valid_before: Timestamp;
  }

interface TokenIssueCsPublicKey {
    cipher: "CS";

    // CS public key.
    cs_pub: Cs25519Point;

    // Start time of this key's validity period.
    valid_after: Timestamp;

    // End time of this key's validity period.
    valid_before: Timestamp;
  }

type ContractTokenDetails =
    | ContractSubscriptionTokenDetails
    | ContractDiscountTokenDetails;

interface ContractSubscriptionTokenDetails {
    class: "subscription";

    // Array of domain names where this subscription
    // can be safely used (e.g. the issuer warrants that
    // these sites will re-issue tokens of this type
    // if the respective contract says so).  May contain
    // "*" for any domain or subdomain.
    trusted_domains: string[];
  };

interface ContractDiscountTokenDetails {
    class: "discount";

    // Array of domain names where this discount token
    // is intended to be used.  May contain "*" for any
    // domain or subdomain.  Users should be warned about
    // sites proposing to consume discount tokens of this
    // type that are not in this list that the merchant
    // is accepting a coupon from a competitor and thus
    // may be attaching different semantics (like get 20%
    // discount for my competitors 30% discount token).
    expected_domains: string[];
  };
```

The wallet must select an exchange that either the merchant accepts directly by listing it in the exchanges array, or for which the merchant accepts an auditor that audits that exchange by listing it in the auditors array.

The Product object describes the product being purchased from the merchant. It has the following structure:

```
interface Product {
    // Merchant-internal identifier for the product.
    product_id?: string;

    // Human-readable product description.
    description: string;

    // Map from IETF BCP 47 language tags to localized
    ↪  descriptions.
    description_i18n?: { [lang_tag: string]: string };

    // The number of units of the product to deliver to the
    ↪  customer.
    quantity?: Integer;

    // Unit in which the product is measured (liters, kilograms,
    ↪  packages, etc.).
    unit?: string;

    // The price of the product; this is the total price for
    ↪  quantity times unit of this product.
    price?: Amount;

    // An optional base64-encoded product image.
    image?: ImageDataUrl;

    // A list of taxes paid by the merchant for this product. Can
    ↪  be empty.
    taxes?: Tax[];

    // Time indicating when this product should be delivered.
    delivery_date?: Timestamp;
  }


interface Tax {
    // The name of the tax.
    name: string;

    // Amount paid in tax.
    tax: Amount;
  }

interface Merchant {
    // The merchant's legal name of business.
    name: string;

    // Label for a location with the business address of the
    ↪  merchant.
    email?: string;

    // Label for a location with the business address of the
    ↪  merchant.
    website?: string;

    // An optional base64-encoded product image.
    logo?: ImageDataUrl;

    // Label for a location with the business address of the
    ↪  merchant.
    address?: Location;

    // Label for a location that denotes the jurisdiction for
    ↪  disputes.
    // Some of the typical fields for a location (such as a street
    ↪  address) may be absent.
    jurisdiction?: Location;
  }

// Delivery location, loosely modeled as a subset of
  // ISO20022's PostalAddress25.
  interface Location {
    // Nation with its own government.
    country?: string;

    // Identifies a subdivision of a country such as state, region,
    ↪  county.
    country_subdivision?: string;

    // Identifies a subdivision within a country sub-division.
    district?: string;

    // Name of a built-up area, with defined boundaries, and a
    ↪  local government.
    town?: string;
    // Specific location name within the town.
    town_location?: string;

    // Identifier consisting of a group of letters and/or numbers
    ↪  that
    // is added to a postal address to assist the sorting of mail.
    post_code?: string;

    // Name of a street or thoroughfare.
    street?: string;

    // Name of the building or house.
    building_name?: string;

    // Number that identifies the position of a building on a
    ↪  street.
    building_number?: string;

    // Free-form address lines, should not exceed 7 elements.
    address_lines?: string[];
  }

interface Auditor {
    // Official name.
    name: string;

    // Auditor's public key.
    auditor_pub: EddsaPublicKey;

    // Base URL of the auditor.
    url: string;
  }

interface Exchange {
    // The exchange's base URL.
    url: string;

    // How much would the merchant like to use this exchange.
    // The wallet should use a suitable exchange with high
    // priority. The following priority values are used, but
    // it should be noted that they are NOT in any way normative.
    //
    // 0: likely it will not work (recently seen with account
    //    restriction that would be bad for this merchant)
    // 512: merchant does not know, might be down (merchant
    //    did not yet get /wire response).
    // 1024: good choice (recently confirmed working)
    priority: Integer;

    // Master public key of the exchange.
    master_pub: EddsaPublicKey;
  }
```

In addition to the fields described above, each object (from `ContractTerms` down) can mark certain fields as "forgettable" by listing the names of those fields in a special peer field `_forgettable`. (See Private order data cleanup.)

B. Contract terms v1 design document

This document was originally written by Christian Grothoff and Florian Dold of the GNU Taler Core Team. While writing this thesis, I added and modified some parts of it. It covers the general design idea of tokens in the GNU Taler payment system.

B.1. Summary

The contract v1 format enables a multitude of advanced interactions between merchants and wallets, including donations, subscriptions, coupons, currency exchange and more.

B.2. Motivation

The existing v0 contract format is too simplistic to support many frequenly requested types of contracts.

B.3. Requirements

We want Taler to support various interesting use-cases:

- Unlinkable, uncopyable subscriptions without accounts (reader can pay with Taler to subscribe to online publication, read unlimited number of articles during a certain period, transfer subscription to other devices, maintain unlinkability / full anonymity amongst all anonymous subscribers).
- Coupons, discounts and stamps – like receiving a discount on a product, product basket or subscription – based on previous purchase(s). Again, with unlinkability and anonymity (modulo there being other users eligible for the discount).
- Subscription tokens lost (due to loss of device without backup) should be recoverable from any previous backup of the subscription.
- Currency conversion, that is exchanging one currency for another.

- Donations, including privacy-preserving tax receipts that prove that the user donated to an entity that is eligible for tax-deductions but without revealing which entity the user donated to. At the same time, the entity issuing the tax receipt must be transparent (to the state) with respect to the amount of tax-deductable donations it has received.
- Throttled political donations where each individual is only allowed to donate anonymously up to a certain amount per year or election cycle.
- Unlinkable gifts – enabling the purchase of digital goods (such as articles, albums, etc.) to be consumed by a third party. For example, a newspaper subscription may include a fixed number of articles that can be gifted to others each week, all while maintaining unlinkability and anonymity between the giver and the recipient.
- Temporally-constrained, unlinkable event ticketing. Allowing visitors to use Taler to purchase a ticket for an event. This ticket grants entry and exit privileges to the event location during a specified time window, while preserving the anonymity of the ticket holder (within the group of all the ticket holders).
- Event deposit systems. A deposit mechanism for events where customers receive a token alongside their cup or plate, which they are expected to return. This system validates that the cup or plate was legitimately acquired (i.e., not brought from home or stolen from a stack of dirty items) and incentivizes return after use.

B.4. Proposed Solution

Merchants will also blindly sign tokens (not coins) to indicate the eligibility of a user for certain special offers. Contracts will be modified to allow requiring multiple inputs (to be *provisioned* to the merchant) and multiple outputs (to be *yielded* by the merchant). The wallet will then allow the user to select between the choices that the user could pay for, or possibly make an automatic choice if the correct choice is obvious. One output option is blindly signed coins from another exchange, possibly in a different currency. Another output option is blindly signed donation receipts from a DONation AUthority (DONAU). Subscriptions can be modeled by requiring the wallet to provision a token of the same type that is also yielded by the contract. For security, payments using subscription tokens (and possibly certain other special tokens?) will be limited to a list of domains explicitly defined as trusted by the token issuer. When paying for a contract, the wallet must additionally sign over the selected sub-contract index and a hash committing it to the blinded envelopes (if any). The merchant backend will (probably?) need to be changed to truly support multiple currencies (ugh).

New Contract Terms Format

The contract terms v1 will have the following structure:

```
interface ContractTermsV1 {
  // This is version 1, the previous contract terms SHOULD
  // be indicated using "0", but in v0 specifying the version
  // is optional.
  version: 1;

  // Unique, free-form identifier for the proposal.
  // Must be unique within a merchant instance.
  // For merchants that do not store proposals in their DB
  // before the customer paid for them, the ``order_id`` can be used
  // by the frontend to restore a proposal from the information
  // encoded in it (such as a short product identifier and timestamp).
  order_id: string;

  // URL where the same contract could be ordered again (if
  // available). Returned also at the public order endpoint
  // for people other than the actual buyer (hence public,
  // in case order IDs are guessable).
  public_reorder_url?: string;

  // Time when this contract was generated.
  timestamp: Timestamp;

  // After this deadline, the merchant won't
  // accept payments for the contract.
  pay_deadline: Timestamp;

  // Transfer deadline for the exchange.  Must be in the
  // deposit permissions of coins used to pay for this order.
  wire_transfer_deadline: Timestamp;

  // Merchant's public key used to sign this proposal;
  // this information is typically added by the backend.
  // Note that this can be an ephemeral key.
  merchant_pub: EddsaPublicKey;

  // Base URL of the (public!) merchant backend API.
  // Must be an absolute URL that ends with a slash.
  merchant_base_url: string;

  // More info about the merchant (same as in v0).
  merchant: Merchant;

  // Human-readable description of the contract.
  summary: string;
```

```
// Map from IETF BCP 47 language tags to localized summaries.
summary_i18n?: { [lang_tag: string]: string };

// URL that will show that the order was successful after
// it has been paid for.  Optional. When POSTing to the
// merchant, the placeholder "${ORDER_ID}" will be
// replaced with the actual order ID (useful if the
// order ID is generated server-side and needs to be
// in the URL).
// Note that this placeholder can only be used once.
// Either fulfillment_url or fulfillment_message must be specified.
fulfillment_url?: string;

// Message shown to the customer after paying for the order.
// Either fulfillment_url or fulfillment_message must be specified.
fulfillment_message?: string;

// Map from IETF BCP 47 language tags to localized fulfillment
// messages.
fulfillment_message_i18n?: { [lang_tag: string]: string };

// List of products that are part of the purchase (see `Product`).
products: Product[];

// After this deadline has passed, no refunds will be accepted.
refund_deadline: Timestamp;

// Specifies for how long the wallet should try to get an
// automatic refund for the purchase. If this field is
// present, the wallet should wait for a few seconds after
// the purchase and then automatically attempt to obtain
// a refund.  The wallet should probe until "delay"
// after the payment was successful (i.e. via long polling
// or via explicit requests with exponential back-off).
//
// In particular, if the wallet is offline
// at that time, it MUST repeat the request until it gets
// one response from the merchant after the delay has expired.
// If the refund is granted, the wallet MUST automatically
// recover the payment.  This is used in case a merchant
// knows that it might be unable to satisfy the contract and
// desires for the wallet to attempt to get the refund without any
// customer interaction.  Note that it is NOT an error if the
// merchant does not grant a refund.
auto_refund?: RelativeTime;
```

```
  // Delivery location for (all!) products (same as in v0).
  delivery_location?: Location;

  // Time indicating when the order should be delivered.
  // May be overwritten by individual products.
  delivery_date?: Timestamp;

  // Nonce generated by the wallet and echoed by the merchant
  // in this field when the proposal is generated.
  // Note: required in contract, absent in order!
  nonce: string;

  // Array of possible specific contracts the wallet/customer
  // may choose from by selecting the respective index when
  // signing the deposit confirmation.
  choices: ContractChoice[];

  // Map from token family slugs to meta data about the
  // respective token family.
  token_families: { [token_family_slug: string]: ContractTokenFamily };

  // Extra data that is only interpreted by the merchant frontend.
  // Useful when the merchant needs to store extra information on a
  // contract without storing it separately in their database.
  extra?: any;

  // Exchanges that the merchant accepts for this currency.
  exchanges: Exchange[];
}

interface ContractChoice {
  // Price to be paid for this choice. Could be 0.
  // The price is in addition to other instruments,
  // such as rations and tokens.
  // The exchange will subtract deposit fees from that amount
  // before transferring it to the merchant.
  amount: Amount;

  // List of inputs the wallet must provision (all of them) to
  // satisfy the conditions for the contract.
  inputs: ContractInput[];

  // List of outputs the merchant promises to yield (all of them)
  // once the contract is paid.
  outputs: ContractOutput[];
  // Maximum total deposit fee accepted by the
  // merchant for this contract.
  max_fee: Amount;
}

type ContractInput =
  | ContractInputRation
  | ContractInputToken;

interface ContractInputRation {
  type: "coin";

  // Price to be paid for the transaction.
  price: Amount;

  // FIXME-DOLD: do we want to move this into a 'details'
  // sub-structure as done with tokens below?
  class: "ration";

  // Base URL of the ration authority.
  ration_authority_url: string;
};

interface ContractInputToken {
  type: "token";

  // Slug of the token family in the
  // 'token_families' map on the order.
  token_family_slug: string;

  // Start of the validity period of the token. This is used to find the
  // matching public key within the token family.
  valid_after: Timestamp;

  // Number of tokens of this type required.
  // Defaults to one if the field is not provided.
  number?: Integer;
};

type ContractOutput =
  | ContractOutputCoin
  | ContractOutputTaxReceipt
  | ContractOutputToken;

interface ContractOutputCoin {
  type: "coins";

  // Amount of coins that will be yielded.
  // This excludes any applicable withdraw fees.
  brutto_yield: Amount;

  // Base URL of the exchange that will issue the
  // coins.
  exchange_url: string;
};

interface ContractOutputTaxReceipt {
  type: "tax-receipt";

  // Base URL of the donation authority that will
  // issue the tax receipt.
  donau_url: string;
};

interface ContractOutputToken {
  type: "token";

  // Slug of the token family in the
  // 'token_families' map on the top-level.
  token_family_slug: string;

  // Start of the validity period of the token. This is used to find the
  // matching public key within the token family.
  valid_after: Timestamp;

  // Number of tokens to be issued.
  // Defaults to one if the field is not provided.
  number?: Integer;
}

type ContractTokenDetails =
  | ContractSubscriptionTokenDetails
  | ContractDiscountTokenDetails

interface ContractSubscriptionTokenDetails {
  class: "subscription";

  // Array of domain names where this subscription
  // can be safely used (e.g. the issuer warrants that
  // these sites will re-issue tokens of this type
  // if the respective contract says so).  May contain
  // "*" for any domain or subdomain.
  trusted_domains: string[];
};

interface ContractDiscountTokenDetails {
  class: "discount";

  // Array of domain names where this discount token
  // is intended to be used.  May contain "*" for any
  // domain or subdomain.  Users should be warned about
  // sites proposing to consume discount tokens of this
  // type that are not in this list that the merchant
  // is accepting a coupon from a competitor and thus
  // may be attaching different semantics (like get 20%
  // discount for my competitors 30% discount token).
  expected_domains: string[];
};

interface ContractTokenFamily {
  // Human-readable name of the token family.
  name: string;

  // Human-readable description of the semantics of
  // this token family (for display).
  description: string;

  // Map from IETF BCP 47 language tags to localized descriptions.
  description_i18n?: { [lang_tag: string]: string };

  // Public keys used to validate tokens issued by this token family.
  keys: TokenIssuePublicKey[];

  // Class-specific information of the token
  details: ContractTokenDetails;

  // Must a wallet understand this token type to
  // process contracts that consume or yield it?
  critical: boolean;

  // Number of tokens issued according to ASS authority
  // FIXME: this is still rather speculative in the design...
  ass?: Integer;

  // Signature affirming sum of token issuance deposit (?) fees
  // collected by an exchange according to the ASS authority.
  // FIXME: this is still rather speculative in the design...
  ass_cost?: Amount;

  // Signature affirming the ass by the ASS authority.
  // FIXME: this is still rather speculative in the design...
  ass_sig?: EddsaSignature;
};


type TokenIssuePublicKey =
  | TokenIssueRsaPublicKey
  | TokenIssueCsPublicKey;


interface TokenIssueRsaPublicKey {
  cipher: "RSA";

  // RSA public key.
  rsa_pub: RsaPublicKey;

  // Start time of this key's validity period.
  valid_after: Timestamp;

  // End time of this key's validity period.
  valid_before: Timestamp;
}


interface TokenIssueCsPublicKey {
  cipher: "CS";

  // CS public key.
  cs_pub: Cs25519Point;

  // Start time of this key's validity period.
  valid_after: Timestamp;

  // End time of this key's validity period.
  valid_before: Timestamp;
}
```

Alternative Contracts

The contract terms object may contain any number of alternative contracts that the user must choose between. The alternatives can differ by inputs, outputs or other details. The wallet must filter the contracts by those that the user can actually pay for, and move those that the user could currently not pay for to the end of the rendered list. Similarly, the wallet must move up the cheaper contracts, so if a contract has a definitively lower price and consumes an available discount token, that contract should be moved up in the list.

Which specific alternative contract was chosen by the user is indicated in the `choice_index` field of the `TALER_DepositRequestPS`.

Output Commitments

When a contract has outputs, the wallet must send an array of blinded tokens, coins or tax receipts together with the payment request. The order in the array must match the order in the outputs field of the contract. For currency outputs, one array element must include all of the required planchets for a batch withdrawal, but of course not the reserve signature.

NOTE: We can probably spec this rather nicely if we first change the batch-withdraw API to only use a single reserve signature.

This array of blinded values is hashed to create the output commitment hash (`h_outputs`) in the `TALER_DepositRequestPS`.

Subscriptions

The user buys a subscription (and possibly at the same time an article) using currency and the contract yields an additional subscription token as an output. Active subscriptions are listed below the currencies in the wallet under a new heading. Subscriptions are never auto-renewing, if the user wants to extend the subscription they can trivially pay for it with one click.

When a contract consumes and yields exactly one subscription token of the same type in a trusted domain, the wallet may automatically approve the transaction without asking the user for confirmation (as it is free).

The token expiration for a subscription can be past the "end date" to enable a previous subscription to be used to get a discount on renewing the subscription. The wallet should show applicable contracts with a lower price that only additionally consume subscription tokens after their end date before higher-priced alternative offers.

Subscription tokens are "critical" in that a wallet implementation must understand them before allowing a user to interact with this class of token. Subscription token secrets should be derived from a master secret associated with the subscription, so that the private keys are recoverable from backup. To obtain the blind signatures, a merchant must offer an endpoint where one can submit the public key of the N-1 subscription token and obtain the blinded signature over the N-th subscription token. The wallet can then effectively recover the subscription from backup using a binary search.

The merchant SPA should allow the administrator to create (maybe update) and delete subscriptions. Each subscription is identified by a subscription label and includes a validity period.

The merchant backend must then automatically manage (create, use, delete) the respective signing keys. When creating an order, the frontend can just refer to the subscription label (and possibly a start date) in the inputs or outputs. The backend should then automatically substitute this with the respective cryptographic fields for the respective time period and subscription label.

Discounts

To offer a discount based on one or more previous purchases, a merchant must yield some discount-specific token as an output with the previous purchase, and then offer an alternative contract with a lower price that consumes currency and the discount token. The wallet should show contracts with a lower price that only additionally consume discount tokens

The merchant SPA should allow the administrator to create (maybe update) and delete discount tokens. Each discount token is identified by a discount label and includes an expiration time or validity duration.

The merchant backend must then automatically manage (create, use, delete) the respective signing keys. When creating an order, the frontend can just refer to the discount token label in the inputs or outputs. The backend should then automatically substitute this with the respective cryptographic fields for the respective discount token.

Donation Authority

A donation authority (DONAU) implements a service that is expected to be run by a government authority that determines eligibility for tax deduction. A DONAU blindly signs tax receipts using a protocol very close to that of the Taler exchange's withdraw protocol, except that the reserves are not filled via wire transfers but instead represent accounts of the organizations eligible to issue tax deduction receipts. These accounts are bascially expected to have only negative balances, but the DONAU may have a per-organization negative balance limit to cap tax deduction receipt generation to a plausible account. DONAU administators are expected to be able to add, update or remove these accounts using a SPA. Tax receipts are blindly signed by keys that always have a usage period of one calendar year.

A stand-alone app for tax authorities can scan QR codes representing DONAU signatures to validate that a given tax payer has donated a certain amount. As RSA signatures are typically very large and a single donation may require multiple blind signatures, CS blind signatures must also be supported. To avoid encoding the public keys, QR codes with tax receipts should reference the DONAU, the year and the amount, but not the specific public key. A single donation may nevertheless be rendered using multiple QR codes.

Revocations, refresh, deposits, age-restrictions and other exchange features are not applicable for a DONAU.

The merchant SPA should allow the administrator to manage DONAU accounts in the merchant backend. Each DONAU account includes a base URL and a private signing key for signing the requests to the DONAU on behalf of the eligible organization.

When creating an order, the frontend must specify a configured DONAU base URL in the outputs. The backend should then automatically interact with the DONAU when the wallet supplies the payment request with the blinded tax receipts. The DONAU interaction must only happen after the exchange confirmed that the contract was successfully paid. A partial error must be returned if the exchange interaction was successful but the DONAU interaction failed. In this case, the fulfillment action should still be triggered, but the wallet should display a warning that the donation receipt could not be obtained. The wallet should then re-try the payment (in the background with an exponential back-off) to possibly obtain the tax receipt at a later time.

Tax Receipts

Tax receipts differ from coins and tokens in that what is blindly signed over should be the taxpayer identification number of the tax payer. The format of the taxpayer identification number should simply be a string, with the rest being defined by the national authority. The DONAU should indicate in its `/config` response what format this string should have, using possibly both an Anastasis-style regex and an Anastasis-style function name (to check things like checksums that cannot be validated using a regex). Wallets must then validate the regex (if given) and if possible should implement the Anastasis-style logic.

Wallets should collect tax receipts by year and offer an export functionality. The export should generate either

- a JSON file,
- a PDF (with QR codes), or
- a series of screens with QR codes.

Wallets may only implement some of the above options due to resource constraints.

The documents should encode the taxpayer ID, the amount and the DONAU signature (including the year, excluding the exact public key as there should only be one possible).

Rationing (future work)

If per-capita rationing must be imposed on certain transactions, a rationing authority (RA) must exist that identifies each eligible human and issues that human a number of ration coins for the respective rationing period. An RA largely functions like a regular exchange, except that eligible humans will need to authenticate directly to withdraw rations (instead of transferring fiat to an exchange). Merchants selling rationed goods will be (legally) required to collect deposit confirmations in proportion to the amount of rationed goods sold. A difference to regular exchanges is that RAs do not charge any fees. RAs may or may not allow refreshing rations that are about to expire for ration coins in the next period.

Once an RA is added to a wallet, it should automatically try to withdraw the maximum amount of ration coins it is eligible for. Available rations should be shown below the subscriptions by RA (if any).

NOTE: RAs are considered an idea for future work and not part of our current timeline.

Limited Donations per Capita (future work)

If per-capita limitations must be imposed on anonymous donations (for example for donations to political parties), an RA can be used to issue donation rations that limit the amount of donations that can be made for the respective period.

NOTE: RAs are considered an idea for future work and not part of our current timeline.

B.5. Definition of Done

- Merchant backend support for multiple currencies
- Merchant backend support for consuming and issuing tokens
- Merchant SPA support for configuring new tokens of different types
- Wallet-core support for various new contract types
- Wallet-core filters for feasible contracts and possibly auto-executes subscriptions
- Wallet-GUIs (WebEx, Android, iOS) render new contract types
- Wallet-GUIs (WebEx, Android, iOS) allow user to select between multiple contracts
- Documentation for developers is up-to-date
- Token anonymity set size (ASS) authority implemented, documented
- Merchants report anonymity set size increases to ASS authority
- Wallets process anonymity set size reports from ASS authority
- Bachelor thesis written on applications and design
- Academic paper written on DONAU (requirements, design, implementation)
- DONAU implemented, documented
- DONAU receipt validation application implemented

- Integration tests exist in wallet-core
- Deliverables accepted by EC

While rationing is part of the design, we expect the actual implementation to be done much later and thus should not consider it part of the "DONE" part. Rationing is complex, especially as a refunded contract should probably also refund the ration.

B.6. Alternatives

The first draft of this DD included the capability of paying with multiple currencies for the same contract (for example, USD:1 and EUR:5) plus tokens and rations. However, this is very complex, both for wallets (how to display), for other merchant APIs (does the refund API have to become multi-currency as well?) and there does not seem to be a good business case for it. So for now, the price is always only in one currency.

B.7. Drawbacks

Significant change, but actually good ratio compared to use-cases covered.

B.8. Discussion / Q&A

(This should be filled in with results from discussions on mailing lists / personal communication.)

C. Project management

To effectively manage the complexities of my solo thesis project, I've implemented a project management framework designed to minimize overhead while ensuring consistent progress and providing valuable feedback on the project's progress. This framework consists of three main pillars.

1. **Comprehensive Task Schedule**: I established a task schedule, visualized through a Gantt chart (see Figure C.1). This schedule outlines the major milestones and tasks of the project. It is segmented into three groups: thesis writing, working on the technical implementation, and other miscellaneous tasks. The timeline of these tasks is updated weekly to stay agile and react to the unpredictable nature of software development.

2. **Weekly Reflections and Planning Sessions**: These sessions form the cornerstone of my ongoing project assessment. After each weekly meeting with my advisor, I review the past week's objectives and evaluate if they were met or not. If the goals are not met, I will reflect on the reasons why and conclude with concrete measures. These insights are documented in the work journal (Appendix D) and are fed back into the continuous adaptation of the task schedule.

3. **Continuous Feedback Loop with Stakeholders**: Weekly meeting with my advisor, Prof. Dr. Christian Grothoff, provide in-depth technical feedback. Additionally, I periodically consult with the expert on this thesis, Han van der Kleij, for a broader perspective feedback on the project's progress and results.

In conclusion, this framework streamlines my workflow and ensures any delays are quickly identified and addressed. Continuous feedback from stakeholders keeps everyone aligned and informed. Moreover, the strategies deployed in this project will be valuable for managing future projects.

C.1. Task schedule

The following figure outlines the main tasks of the project split into three groups: thesis writing, technical implementation and miscellaneous tasks. As mentioned in the chapter introduction, the schedule gets updated continuously to reflect the current state of the project.

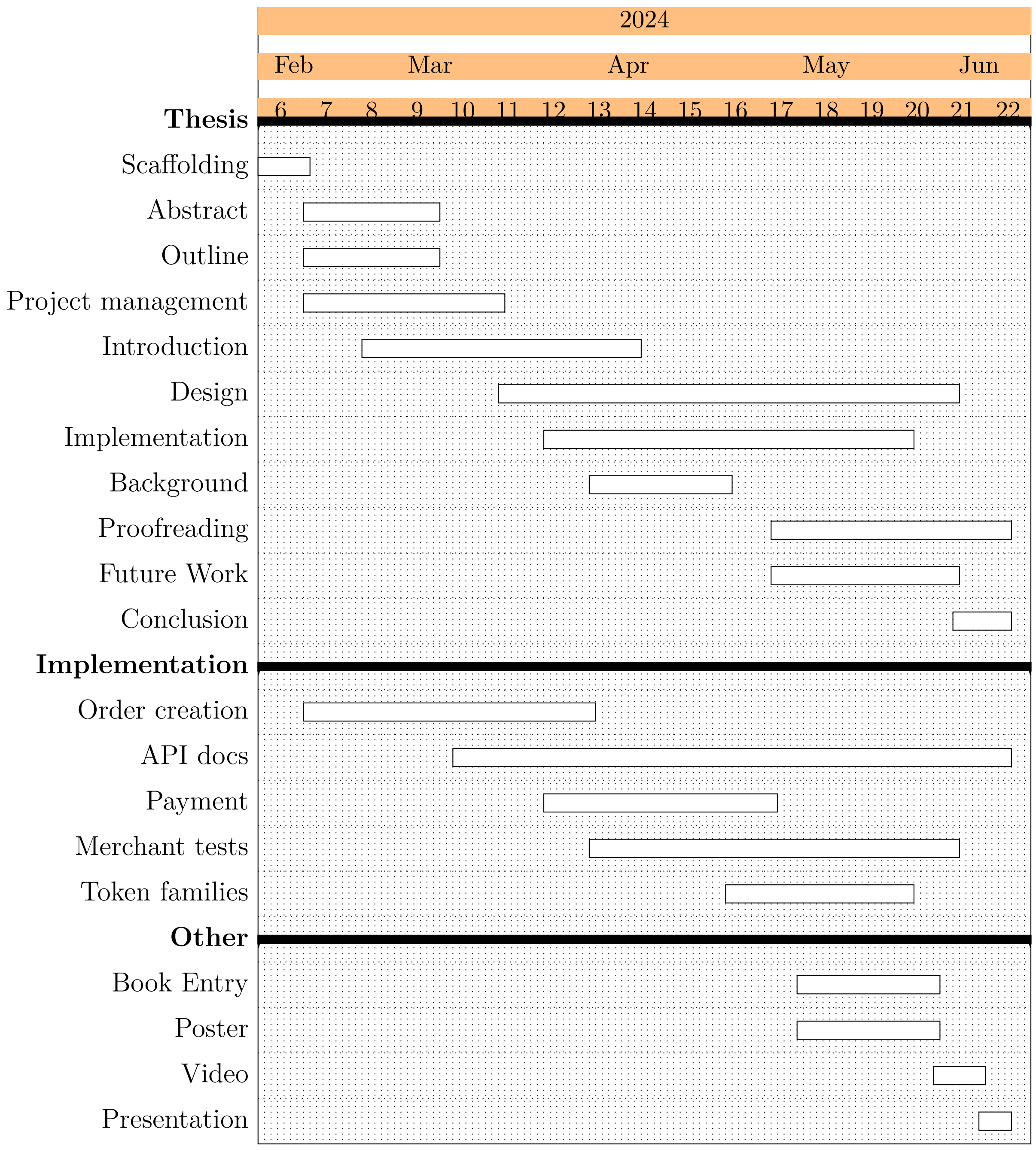


Figure C.1: Task schedule overview of the thesis project.

C.2. Goals

The success of this thesis hinges on achieving a set of structured goals, defined in the initial meeting with my advisor. These goals are categorized into three escalating tiers, each representing a higher level of achievement and contribution to the GNU Taler project. The categorization serves as the groundwork for a clear roadmap and is used to measure the outcome of the project.

1. **Baseline goal** *(Minimum requirements to achieve the project's basic purpose):*
   The specification of the new contract version (v1) is finalized to support subscriptions and discounts. The merchant backend is extended to support orders with the v1 contract format. Any API changes are documented. Existing tests are still passing and new tests were created to cover the extended functionality.

2. **Core goal** *(Beyond basic requirements, making significant contributions to the GNU Taler project):*
   In addition to the baseline goal, subscriptions and discounts are also supported in wallet-core. If applicable, tests were created to cover new functionality.

3. **Stretch goal** *(Pushing the boundaries of the thesis, fostering innovation, and leading to exceptional outcomes):*
   Subscriptions and discounts are supported in at least one wallet. The solution was integrated into the online-publishing platform of WOZ Die Wochenzeitung to support newspaper subscriptions. Within a test/staging environment, customers can buy subscriptions and use them to read articles. Implementation into the different wallets may be delegated to members of the GNU Taler project.

The defined goals allow for some flexibility in the planning of the thesis. If certain tasks unexpectedly take longer, they allow to reduce the scope without sacrificing core functionality. While the stretch goal also allows to invest more work to reach an ambitious goal.

D. Work journal

Meeting 22.2.2024

Present at the meeting were: Christian Grothoff and Christian Blättler. The meeting took place online.

Summary

As the first meeting for this thesis, the main aim of the meeting was to agree on the goals for the thesis project. Also, we coordinated the planning of the meetings with thesis expert Han van der Kleij, communicated our expectations at the project and talked about all the deliverables.

Next Steps

The next meeting will take place online on 28.2.2024. Until then I will start work at the merchant, parsing the v1 contract terms. I'll also setup the git repository required for the thesis and outline the contents.

Meeting 28.2.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

We discussed my draft of the data structures to parse the new version of the contract terms found at `https://docs.taler.net/design-documents/046-mumimo-contracts.html`. I was able to collect fruitful feedback of the two professors. We also briefly talked about the project management of the thesis and how my progress of the written part was. I pointed out that I created a git repository with the latex project for my thesis and I started on the abstract as well as the work journal.

Next Steps

The next meeting will take place online on 6.3.2024. Until then I will continue my work on the post orders endpoint in the merchant backend. I will create data structures and methods to handle the parsing of the contract terms in v0 and v1. Additionally I will create methods that parse and serialize the contract terms stored in the database.

Meeting 6.3.2024

Present at the meeting were: Christian Grothoff, Han van der Kleij, Emmanuel Benoist, Florian Dold and Christian Blättler. The meeting took place online.

Summary

This was the first meeting where I got to know the Expert on my thesis, Mr. van der Kleij. I started by introducing myself and giving a brief introduction into the Taler project. Thankfully Mr. van der Kleij was already familiar with the Taler project, due to previous theses he accompanied. After Mr. van der Kleij introduced himself, he informed me about some organizational matters. He'd like to get the thesis submitted to him as a PDF and a link to the code in a git repository.

Next, I presented an overview of the thesis, our previous work and our work methodology. This transitioned into questions from Mr. van der Kleij regarding project management, testing and UX.

After an hour of questions, answers and discussions, we agreed to meet again after the bulk of the work on the merchant is finished.

Next Steps

The next meeting will take place online on 13.3.2024. Since there was no time left for technical questions in this meeting, I will ask them in the next meeting. I will keep working on the orders creation handler.

Meeting 13.3.2024

Present at the meeting were: Christian Grothoff and Christian Blättler. The meeting took place online.

Summary

I brought up some suggested changes to the v1 contract specifications. Specifically I'd wasn't sure about the use of the `limits` interface in the new contract terms. Also it was unclear to me how I should handle the concept of rationing that is mentioned in the design document. After some discussion we decided to drop the suggested `limits` interface and revert back to the `max_fee` field. Also I was told to ignore the idea of rationing for now in my implementation.

Furthermore we decided that a choice only consists of inputs and outputs and the products array should stay the same for all choices.

Next Steps

I will work on the validation of input and outputs next. Every token referenced in a choice has to be validated for existence and the corresponding metadata has to be fetched from the database.

Meeting 20.3.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

I noted that I moved the `token_types` field, that was previously on the contract choice, up of the contract terms in the specification of the design document. Also we agreed on using the token family slug to identify a token family in the API. The client using the API should need to have no knowledge of the cryptography used for tokens. This means no keys or cipher (RSA, CS) fields should be used in the API.

I also started a discussion on how metadata of tokens should be stored in the contract terms table of the database and how much the backend would fetch from the token family database table when a wallet wants to pay for an order. We agreed upon storing all data relevant to the wallet in the contract terms table, since this data is essentially 'frozen in time' and should not change if a token family is edited.

Next Steps

Next I plan to get the test running (make check). I had some issues connecting to the database, using the correct database schema previously. Also I believe that sometimes I encounter a race condition, since my containerized test environment seems to have higher latencies for some tasks.

Meeting 27.3.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

In this meeting we decided upon renaming `token_authority_label` to `token_slug` to be consistent with the token families API. Also we decided to use the base URL of the merchant backend as the default value for the trusted domains of a token and that the addition of the `valid_after` timestamp is mandatory for all inputs and outputs, since this field, in conjunction with the token family slug is required for the backend to choose the correct signing key.

We had a discussion about how the start date of a newly generated token signing key should be calculated. We found, that it makes sense to have a rounding duration field on the token family, that is configurable individually for each token family. We also briefly talked about how the pay endpoint of the merchant API has to be changed to support multiple choices in a contract and how backup of tokens could work from the perspective of customers.

I got feedback on the outline of the written thesis. We agreed upon switching some sections between chapters.

Next Steps

We will have the next meeting in 14 days. Until then I want to get a 'happy path' example working for the order creation endpoint. The plan is to have a simple cURL request that creates an order with input and output tokens. The backend then has to check if valid keys already exist for the given token and if not, generate keys for the configured cipher.

Meeting 10.4.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

Before this meeting I specified how public keys for token families should be stored in the database. We agreed that my suggestion works as intended. We also discussed on which verbs we want to use to refer to interactions with tokens and landed upon the following. A merchant *issues* tokens that can then be *used* by a customer.

I decided to drop the naming of 'token authority' completely and use token family throughout the specification.

Also recognized that I'm currently behind schedule and planned to delegate some wallet related implementation tasks to other members of the GNU Taler team. After the 0.10.0 release of GNU Taler, I will merge the current state of my code into the main branch so other team members can develop against it. But before I can do that, I have to add a field to payed orders that indicates which choice the customer selected upon payment.

Dr. Benoist reminded me, that in the midst of implementation work, I should not forget writing the actual thesis.

Next Steps

In the next seven days I'll focus on writing up the design chapter of my thesis. Furthermore I'm planning to add a field to the order details endpoint that indicates the selected choice for paid orders.

Meeting 17.4.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

Over the past week, I made significant progress on both the written and practical parts of my thesis. On the written part, I wrote the first draft of the design chapter. This involved outlining the core concepts and architectural decisions. To supplement the written content, I added some diagrams and illustrations. I also made sure to bring the work journal up to date with all the relevant work done.

Regarding the implementation, I focussed on writing tests. First I expanded the merchant library with the functionality to create a new token families. This feature was then used in a new testing command used to create token families. I also added a new testing command to create v1 orders that are using tokens. I also added a basic cURL test covering v1 orders and token families to `test_merchant_order_creation.sh`.

During the meeting I got feedback from Prof. Dr. Benoist regarding the progress of the written part of my thesis. He mentioned that the transition into to the design chapter is bit abrupt, so I will add a high-level introduction, before diving into the juicy details.

We also discussed where in the code-flow of the payment handler I should add the token validation. After some back and forth, we concluded that it's best to first validate all token-related input, before continuing with the validation of coins. This allows us to abort pay requests with invalid token-related input earlier, without putting additional strain on the central exchange service by doing coin validation.

Next Steps

The next steps consist of the following points.

- Merge my current working branch into the main branch of the respective git repositories.
- Create a new branch to start work on the processing of tokens in the payment verification process.
- Add a new field to the `merchant_contract_terms` database table to store the order choice a customer selected.
- Query the newly added field and return it as part of the order details endpoint of the merchant RESTful API.
- Extend the payment endpoint of the merchant RESTful API as specified in the documentation.
- Implement token validation on the payment handler.

Meeting 24.4.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

In the time since the last meeting I merged my current working branches and created new ones for the changes regarding the payment handler. I also added the a field `choice_index` to the contract terms database table of the merchant backend. This field is being returned when the back office requests details of a paid order.

I started work on supporting input tokens in the payment POST endpoint. The backend will no parse and validate tokens provided as inputs. It also fetches the necessary keys and contract information from the database. For the time being, I added placeholder functions for validating all token-related signatures, so I don't have to worry about the crypto design details for now.

I also worked on the rounding logic for token start dates. My goal was to create a rounding logic that is easily understandable and predictable by humans. Dates can be a complex topic on their own, so I will definitely write-up a section in the thesis about this.

During the meeting we also discovered interesting use-cases regarding the intersection of tokens and DONAU (donation authority). For example if you become a Rega[1] patron, you're eligible to deduct this payment from your taxes, but at the same time you could receive a token that proofs that your a patron for the current calendar year.

[1] `https://www.rega.ch/en/rega-patron/become-a-patron/your-benefits-as-a-rega-patron`

Or consider paying at a supermarket where you can round up the total amount and donate that to charity. You'd receive your loyalty stamps as tokens and also a tax receipt to deduct the amount you donated. I will document these use-cases in the thesis, to highlight the flexible design and why this is a requirement for many use-cases.

We also discussed where in the payment validation flow the merchant backend should sign newly issued output tokens. We decided, that it's best for the merchant backend to sign output tokens (token envelopes) *before* sending the network request to the exchange. This way the outputs are already signed and ready as soon as the payment confirms. This ensures that any errors during the token issuing process are discovered before the payment is confirmed. This decision requires a new table to be added to the merchant database. Blindly signed output tokens will be stored in this table. Once the exchange confirms the deposit of coins, the merchant backend can simply select the previously signed tokens from the database and return them to the wallet.

Next Steps

The next steps consist of the following points.

- Design data structures for token use signatures made by the wallet.
- Adding a section about the cryptography design to the thesis.
- Overhaul the introduction into the design chapter. It should be a simpler introduction, focussing on explaining all the actors (merchant, customer) and entities (contract terms, token family) in the system. Explain the communication and add a diagram to illustrate it.
- Writing a subsection about the token duration rounding in the design chapter of the thesis.
- Weave the newly discovered use-cases into the content of the thesis. Emphasize how the design of the solution allows for such use-cases and that they are quite common.
- Implement token use signing and signature validation in merchant backend.
- Write a test case to cover the payment of an order with input tokens.
- Add a table `merchant_issued_tokens` to the merchant database to store the signed tokens before they are returned to the wallet.

Meeting 24.4.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

In the last seven days I focussed on getting the merchant backend in a state where it validates input tokens and issues output tokens into token envelopes provided by the wallet. This was a quite an involved task that took hours of work, but I got it there in the end.

Since this week was focussed on the technical implementation, I will use the next seven days mostly to work on the other deliverables such as the written thesis, book entry and poster. I will also work out some wrinkles and pending todos in the code.

During the meeting we agreed that token envelopes should not include the hash of the corresponding public key, but rather follow an ordering convention to be matched with the issue public keys. This will simplify the surface of the RESTful API of the merchant backend, but require an additional section on the ordering convention. We also agreed that 128 is a good limit for number of tokens per order.

Also the merchant backend needs a new endpoint (`/csr-token`) to support tokens using Clause Schnorr signatures[2]. The endpoint will have the hash of the issue public key and a wallet-generated nonce as inputs and return values required for the blinding of CS token envelopes. It's almost identical to the `/csr-withdraw` endpoint of the exchange[3].

Next Steps

The next steps consist of the following points.

- Add a table `merchant_issued_tokens` to the merchant database to store the signed tokens before they are returned to the wallet. *(Copied from the week before)*
- Adding a section about the cryptography design to the thesis. *(Copied from the week before)*
- Draft a first version of the poster.
- Work on the book entry. It should be accessible and understandable for people without deep knowledge of cryptography or software engineering.
- Overhaul the introduction into the design chapter. It should be a simpler introduction, focussing on explaining all the actors (merchant, customer) and entities (contract terms, token family) in the system. Explain the communication and add a diagram to illustrate it. *(Copied from the week before)*
- Writing a subsection about the token duration rounding in the design chapter of the thesis. *(Copied from the week before)*
- Weave the newly discovered use-cases into the content of the thesis. Emphasize how the design of the solution allows for such use-cases and that they are quite

[2] `https://taler.net/papers/cs-thesis.pdf`

[3] `https://docs.taler.net/core/api-exchange.html#post-csr-withdraw`

common. *(Copied from the week before)*

Meeting 1.5.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

In the last seven days I reached a milestone, as the tests for buying and using an RSA-based token passed for the first time. Some important checks are still missing when using a token, but the general concept works. To get these tests to pass, I had to implement the EdDSA signing and RSA logic of the wallet in the test lib. I also had to add new traits for sharing tokens between different test cases. Besides the technical implementation, I also updated the working journal.

During the meeting, we decided that I would focus on the deliverables, such as the poster and the book entry, before continuing with the technical implementation. My advisors also gave feedback on the design chapter of the thesis. I will revise the chapter introduction and draw some schematic flows that illustrate the idea of token-based subscriptions compared to traditional solutions.

Next Steps

As mentioned before, I will focus on the deliverables, such as the poster, the book entry, and the written thesis. Regarding the thesis, the implementation chapter will be my main focus in the coming weeks.

There are still open tasks regarding the technical implementation, but none that I don't know how to solve. Therefore, I will focus on getting the thesis in a state that is ready for a first round of feedback, and then finish the technical implementation while waiting for the feedback.

The next meeting will take place on May 15th, since both of my advisors will not be available on May 7th.

Meeting 15.5.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

It had been fourteen days since the last meeting, and I used that time to draft a first version of the book entry and poster. I also designed a schematic flowchart for the purchase and use of a token-based subscription that I will use in the book entry, poster, and written thesis document. Additionally, I've written a first big part of the implementation chapter. Finally, the team doing their bachelor thesis on tax-deductible donations (DONAU) contacted me and I implemented a JSON helper method in the GNUnet repository for them.

During the meeting I asked for feedback on the book entry and the poster. We also discussed the use case for free trails that my work colleagues brought up, and concluded that free trails do not work with the unlinkability and anonymity properties. We also talked about how the wallet would derive token private keys and the possibility of backups, and decided that this should be a topic for the discussion chapter of the thesis.

Next Steps

I will add a QR code to the poster and book entry that points to a news article on the GNU Taler website. This news article is located in the `/template/news/2024-09.html` directory of the Taler website git repository. I will also add more graphics to the poster and probably switch from Microsoft PowerPoint to Latex for the layout. The goal for the next meeting is to implement the feedback provided for the book entry and the poster. I also want to finish the draft of the implementation chapter so I can get feedback on it.

Meeting 22.5.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

Last week I implemented the feedback for the book entry and the poster. Before the meeting, I sent them both to my advisor so that we could use the meeting to discuss them. We used the whole hour to discuss the content of the book entry and the poster. We also talked briefly about the dissertation, and I got some feedback on the introduction and design chapters. We concluded that the strongest argument for privacy-preserving subscriptions is to prevent the extraction of sensitive information from personalized profiles, especially for people in regions with repressive regimes.

Next Steps

Over the next seven days, I will go over the entire thesis again, with the goal of improving the flow and wording of all chapters. In addition, I have scheduled a meeting with the thesis expert on May 23rd to present the current progress and discuss any questions.

Meeting 29.5.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

During the last seven days I have been working through the whole thesis to get it ready for the next round of feedback. Christian Grothoff suggested to send this draft to some working groups within the GNU Taler project for feedback, which I agreed to do. Emmanuel Benoist mentioned that some sample interactions with the merchant RESTful API would be helpful, so I will add those before sending the thesis off for review. Christian Grothoff also brought up the idea of a deployment option of the solution that is less privacy-focused but more pragmatic: instead of using a token for every single subscription good you consume, the customer would use a token to buy a short-lived (30-60min) session cookie. This cookie can then be used to consume subscription goods for that time period. This option would reduce the complexity of using a subscription, but still have reasonable protection against profiling and subscription sharing due to the short lifetime of the tokens.

Next Steps

The next steps consist of the following items.

- Add sample interactions (using CURL) to the Merchant API section of the design chapter.
- Add a section proposing the idea of using tokens to purchase session cookies.
- Prepare thesis for review: address outstanding TODOs and feedback.
- Submit thesis for review.

Meeting 5.6.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

In the days leading up to this meeting I added an examples section to the API section of the design chapter. It shows how CURL can be used to do different API request. Additionally I added a section in the discussion chapter that proposes the idea of using tokens to buy session cookies. I submitted the thesis for a review round and already got to implement some of the feedback I received.

During the meeting we discussed some feedback on the newly added API examples.

We decided to remove the –verbose flag and use backend.demo.taler.net as the test host. I will also clarify that the design document in the appendix was mostly authored by Florian Dold and Christian Grothoff and the API docs by the GNU Taler community.

Next Steps

The next steps consist of the following items.

- Make sure API docs (`https://docs.taler.net/`) match with thesis.
- Implement improvements and remaining feedback.
- Contact admin@taler.net to setup christian.blaettler@taler.net email address.
- Mention Migros and Coop as examples for loyalty programs.
- Write news article for `https://taler.net/en/news/`.
- Produce video.

Meeting 12.6.2024

Present at the meeting were: Christian Grothoff, Emmanuel Benoist and Christian Blättler. The meeting took place online.

Summary

In the past seven days I produced and submitted the video.

This was our last meeting. We discussed some last feedback for the thesis. Christian Grothoff suggested to add a conclusion chapter to discuss the bigger picture that this thesis lives in. Furthermore, Emmanuel Benoist suggested to add a discussion section about what happens when a subscription is lost or stolen.

Last Steps

The last steps consist of the following items.

- Write conclusion.
- Add section about termination of subscriptions.
- Ensure code is consistent with API docs and thesis.
- Merge code to master.
- Import relevant endpoints of API docs to appendix.
- Import design document to appendix.
- Submit thesis. :-)