MODERN TREASURY

Accounting for Developers

Accounting For Developers

In this ebook, we walk through basic accounting principles for anyone building products that move and track money.

Introduction

As a payment operations startup, accounting principles are core to our work. We have seen them implemented at scale at some of the largest fintechs and marketplaces. Yet, accounting seems like an arcane topic when you're starting out. This HackerNews thread, for example, is rather representative of the state of confusion around the topic.

Over the years, there have been many accounting for developer guides published (two great ones are Martin Blais' and Martin Kleppmans'), but we wanted to add our two cents to the discussion. In our experience, a concepts-first approach to explaining accounting comes in handy when you are designing systems that move or touch money.

In this ebook, we will cover the foundational accounting principles, then bring it all together by walking through how to build a Venmo clone.

Is this guide for you?

This guide is designed for developers that work on applications that handle money in any way. You may work at a fintech company (the data you handle is money), or perhaps you are responsible for managing the fintech integrations in your startup (you handle data and money). We think that every engineer that builds or maintains such systems benefits from knowing the core principles of accounting.

CHAPTER 1

Does Accounting Really Matter In Software Development?

Double-entry systems are more reliable at tracking money than any other viable alternative. As a payments infrastructure company, we often get to see the architecture of some of the most successful software companies. One design is constant: they use double-entry accounting in their code. Some build their applications from the start with accounting concepts in mind, but in most cases, companies begin incorporating these concepts only after their original code starts causing problems.

When software fails to track money properly, it does so in a number of common patterns. The most common failure mode is software accidentally creating or destroying records of funds. This leads to all sorts of inconsistencies. Every developer we know has horror stories about explaining to their finance team why a customer is owed money or what caused a payout to have an unexpected amount. Internal records differing from bank statements, reconciliation engines gone awry, balances that don't make sense given a set of transactions—these are all problems that can be mitigated with double-entry accounting. For more evidence that double-entry systems are a good standard for scalable applications, see the stories of Uber, Square, and Airbnb.

The core principle of double-entry accounting is that **every transaction should record both where the money came from and what the money was used for**. This guide explains why that is and how it works.

CHAPTER 2

The Building Blocks Of An Accounting System

A good starting point is understanding accounts and transactions.

Accounts

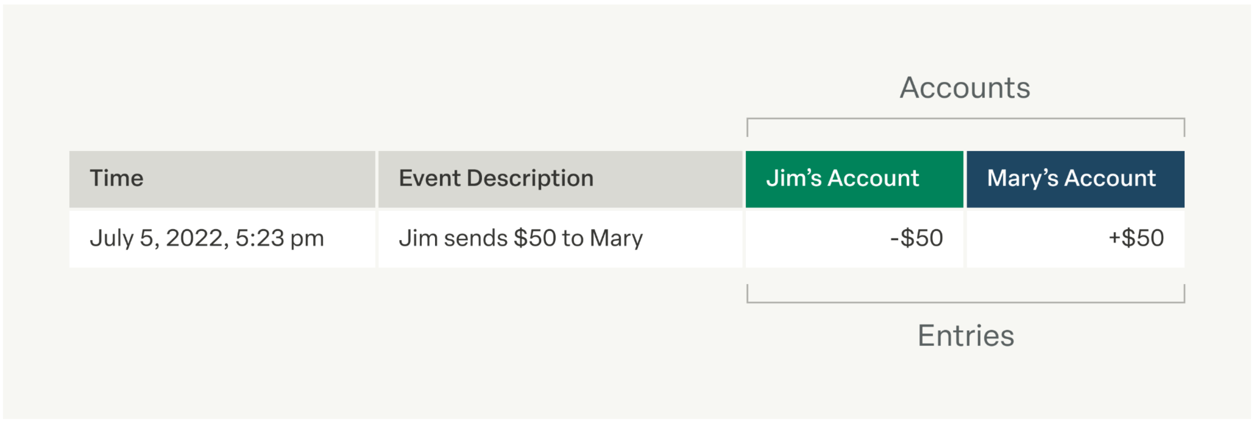
An account is a segregated pool of value. The easiest analogy here is your own bank checking account: money that a bank is holding on your behalf, clearly demarcated as yours. Any discrete balance can be an account: from a user's balance on Venmo to the annual defense spending of the United States. Accounts generally correlate with the balances you want to track.

In accounting, accounts have *types*. More on this later.

Transactions

Transactions are atomic events that affect account balances. Transactions are composed of entries. A transaction has at least two entries, each of which corresponds to one account.

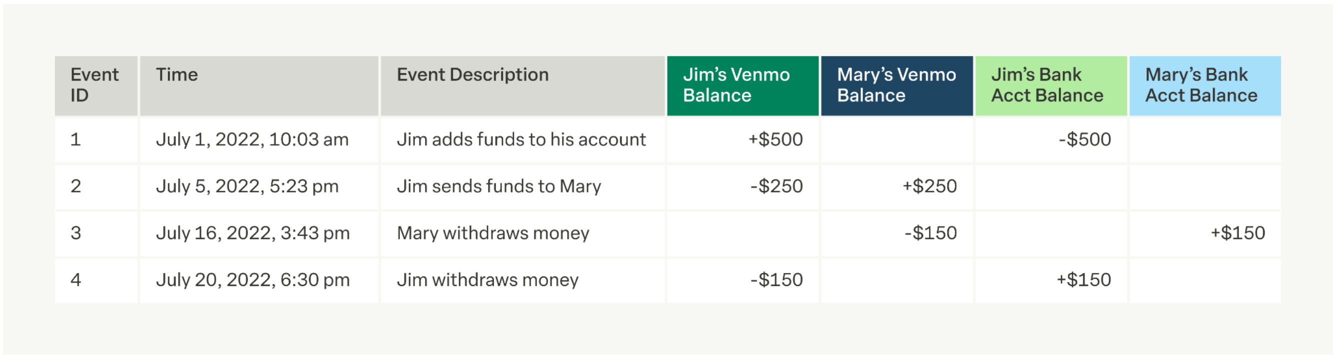
Let's use a simple Venmo transfer as an example. Jim is sending $50 to Mary:


Accounts
Entries

Time	Event Description	Jim's Account	Mary's Account
July 5, 2022, 5:23 pm	Jim sends $50 to Mary	-$50	+$50

The entries in this transaction tell which accounts were affected. If each user's balance is set up as an account, a transaction can simultaneously write an entry against each account.

Now, let's expand this model with more accounts and additional events:



Event ID	Time	Event Description	Jim's Venmo Balance	Mary's Venmo Balance	Jim's Bank Acct Balance	Mary's Bank Acct Balance
1	July 1, 2022, 10:03 am	Jim adds funds to his account	+$500		-$500	
2	July 5, 2022, 5:23 pm	Jim sends funds to Mary	-$250	+$250		
3	July 16, 2022, 3:43 pm	Mary withdraws money		-$150		+$150
4	July 20, 2022, 6:30 pm	Jim withdraws money	-$150		+$150	

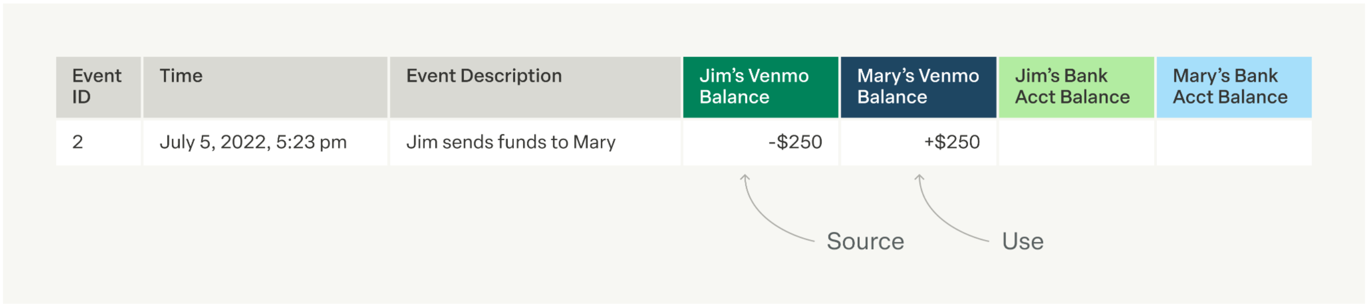
Simple ledger of Jim and Mary's accounts

Here I have a ledger—a log of events with monetary impact. We often see developers mutating balances directly rather than computing a balance from a log of transactions. This is suboptimal.

While mutating a balance directly is more efficient and simpler to implement, it's more accurate to store immutable transactions and always compute balances from those transactions. Mutating balances creates a system that is prone to errors, as it becomes non-trivial to detect and reconcile inaccuracies.

Notice how each transaction has multiple entries. Each entry belongs to a transaction and an account. By comparing entries side by side, one can understand where the money came from and what it was used for.

Double-entry ensures that, as transactions are logged, sources and uses of funds are clearly shown, and balances can be reconstructed as of any date:



Event ID	Time	Event Description	Jim's Venmo Balance	Mary's Venmo Balance	Jim's Bank Acct Balance	Mary's Bank Acct Balance
2	July 5, 2022, 5:23 pm	Jim sends funds to Mary	-$250	+$250		

Zooming in on a transaction from Jim to Mary

This core idea—**one transaction, at least two entries, one representing the source and the other representing the use of funds**—is one of the foundational ideas of double-entry accounting. We'll expand more on this later.

Dual Aspect

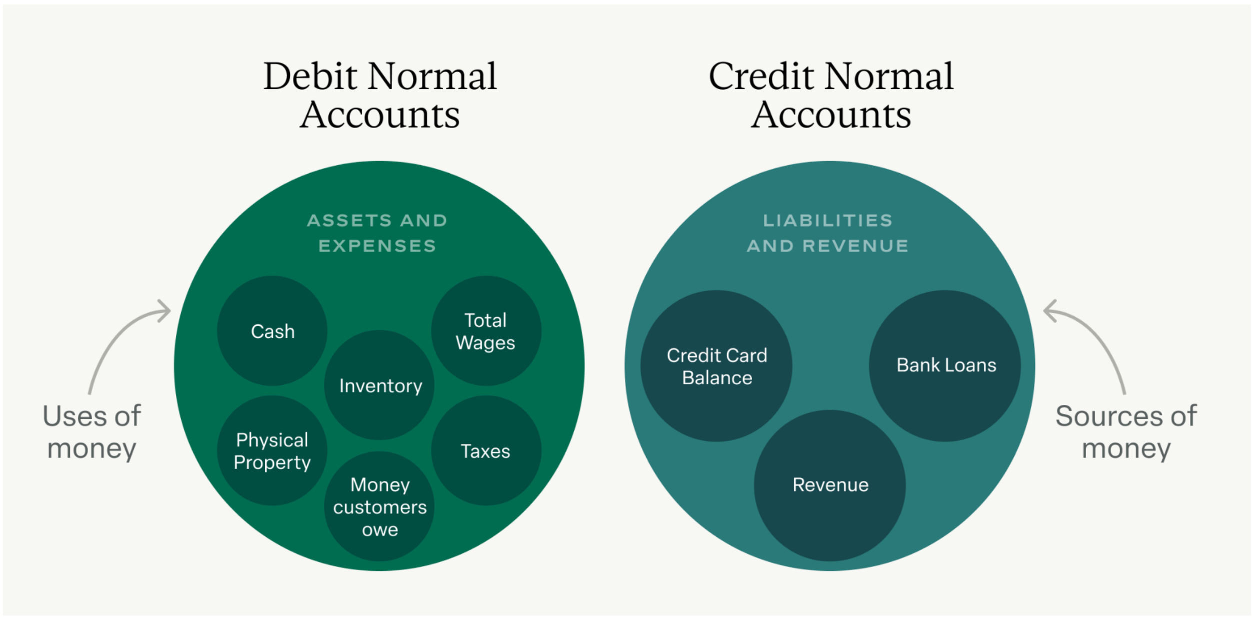
As mentioned, another big innovation of accounting was creating account types. The two types we will cover here are debit normal and credit normal:

- Accounts that represent funds you own, or uses of money, are debit normal accounts.
- Accounts that represent funds you owe, or sources of money, are credit normal accounts.

Let's illustrate: the right side lists credit normal accounts and the left side lists debit normal accounts. We'll put accounts tracking **uses** of funds on the debit normal side and accounts tracking **sources** of funds on the credit normal side.

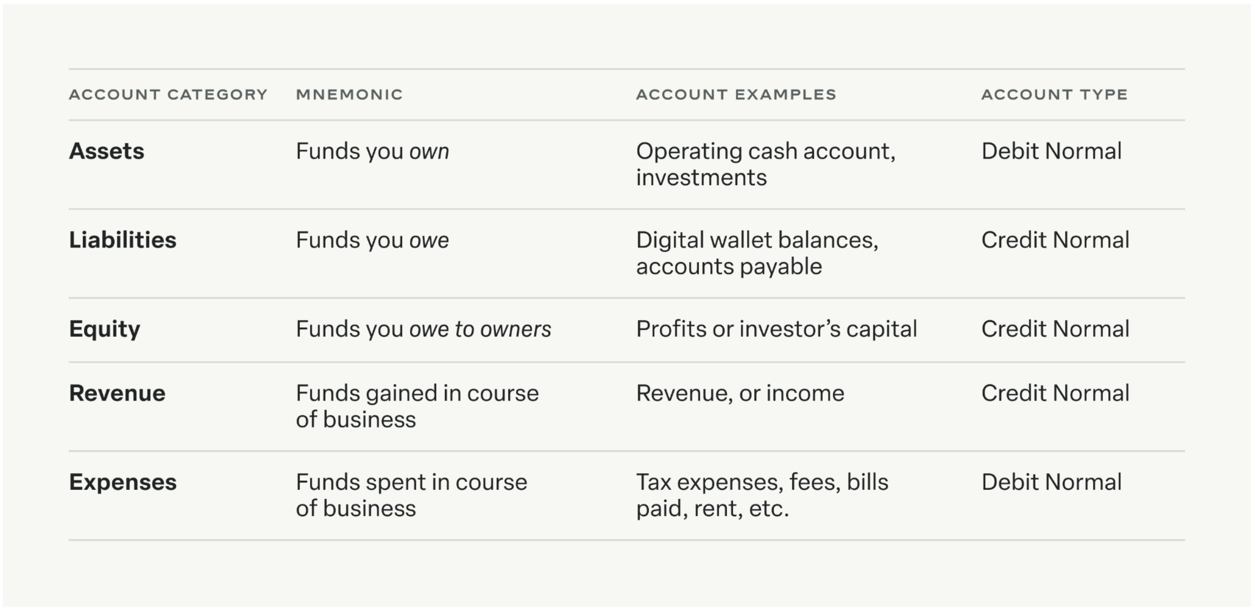
Examples of **uses** of funds are **assets** and **expenses**. Buying inventory, making an investment, acquiring physical property, and so on. The term "use" is broadly defined: letting cash sit in a bank account is a use of funds, as well as selling on credit to someone else (you are effectively "using" the money you'd get on a sale by extending them credit). The accounts that represent these balances are all debit normal accounts.

Conversely, **sources** of funds—**such as liabilities, equity, or revenue**—can mean bank loans, investors' capital, accumulated profits, or income. "Source" is broadly defined, too: if you are buying on credit, for instance, that is a "source" of money for you in the sense that it prevents you from spending money right now. Accounts that represent these balances are credit normal accounts.



What ladders up to debit normal and credit normal accounts

Here is a handy table with the different account types:



ACCOUNT CATEGORY	MNEMONIC	ACCOUNT EXAMPLES	ACCOUNT TYPE
Assets	Funds you *own*	Operating cash account, investments	Debit Normal
Liabilities	Funds you *owe*	Digital wallet balances, accounts payable	Credit Normal
Equity	Funds you *owe to owners*	Profits or investor's capital	Credit Normal
Revenue	Funds gained in course of business	Revenue, or income	Credit Normal
Expenses	Funds spent in course of business	Tax expenses, fees, bills paid, rent, etc.	Debit Normal

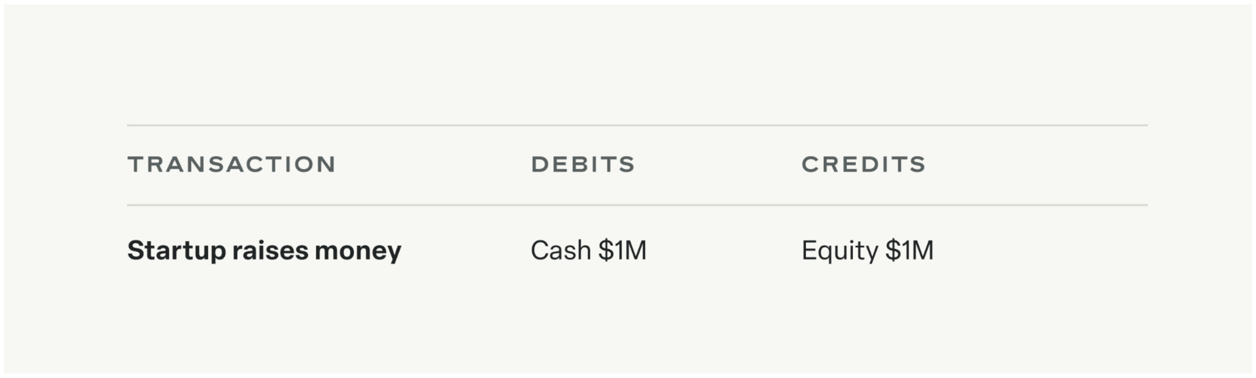
Account categories with simple mnemonics and examples

(Notice that debit cards hold money you own, while credit cards hold money you owe.)

Debits and credits

Some of the guides we mentioned at the beginning of this post advise developers to “save the confusion and flush out debits and credits from your mind.” We do recognize that debits and credits can be challenging to grasp, but we think fully mastering these concepts is important when creating transaction handling rules.

Part of the confusion is that “debits” and “credits” are often used as verbs: to debit or to credit an account. Debits and credits can also refer to entries, for example:



TRANSACTION	DEBITS	CREDITS
Startup raises money	Cash $1M	Equity $1M

Debit and credit entries

This sample transaction has two entries: we’re **debiting** our cash account and **crediting** our equity account for $1M. Save for a few special situations, accounting systems only log positive numbers. The effect on balances will depend on whether the entry is on the “debit side“ or “credit side.”

Debits and credits are a shorthand for the expected effects on accounts, depending on their type. A credit entry will always increase the balance of a credit normal account and decrease the balance of a debit normal account. Put differently:

ENTRY	DEBIT NORMAL ACCOUNTS	CREDIT NORMAL ACCOUNTS
Debit	Increase balance	Decrease balance
Credit	Decrease balance	Increase balance

Entries and their expected effects

Let's model out a few transactions to drive this point home. Let's use a fictitious startup called Modern Bagelry—an eCommerce store for premium bagels.

In this example, we will use four accounts: **Cash** and **Inventory** (both debit normal accounts) as well as **Equity** and **Loans** (both credit normal accounts). Let's say this ledger starts on a day T, and we are measuring time in days.

TIME	TRANSACTION	DEBITS	CREDITS	EXPLANATION
T	Modern Bagelry raises money	Cash $1M **Increase**	Equity $1M **Increase**	MB (Modern Bagelry) raises $1M in initial capital. This increases both **Cash** and **Equity** balances.
T+1d	Modern Bagelry buys inventory	Inventory $300k **Increase**	Cash $300k **Decrease**	MB then buys $300k in inventory. This increases the **Inventory** account but decreases the **Cash** account.
T+3d	Modern Bagelry sells inventory	Cash $50k **Increase**	Inventory $50k **Decrease**	MB proceeds to sell $50k in inventory. This decreases the **Inventory** balance but increases **Cash**.
T+5d	Modern Bagelry takes a loan	Cash $500k **Increase**	Loans $500k **Increase**	To deal with future expenses, MB then takes a $500k loan. This increases the **Loans** balance, as well as the **Cash** balance.
T+10d	Modern Bagelry pays loan installment	Loans $30k **Decrease**	Cash $30k **Decrease**	Finally, MB pays $30k in loan installments. This decreases balances for both **Cash** and **Loan** accounts.

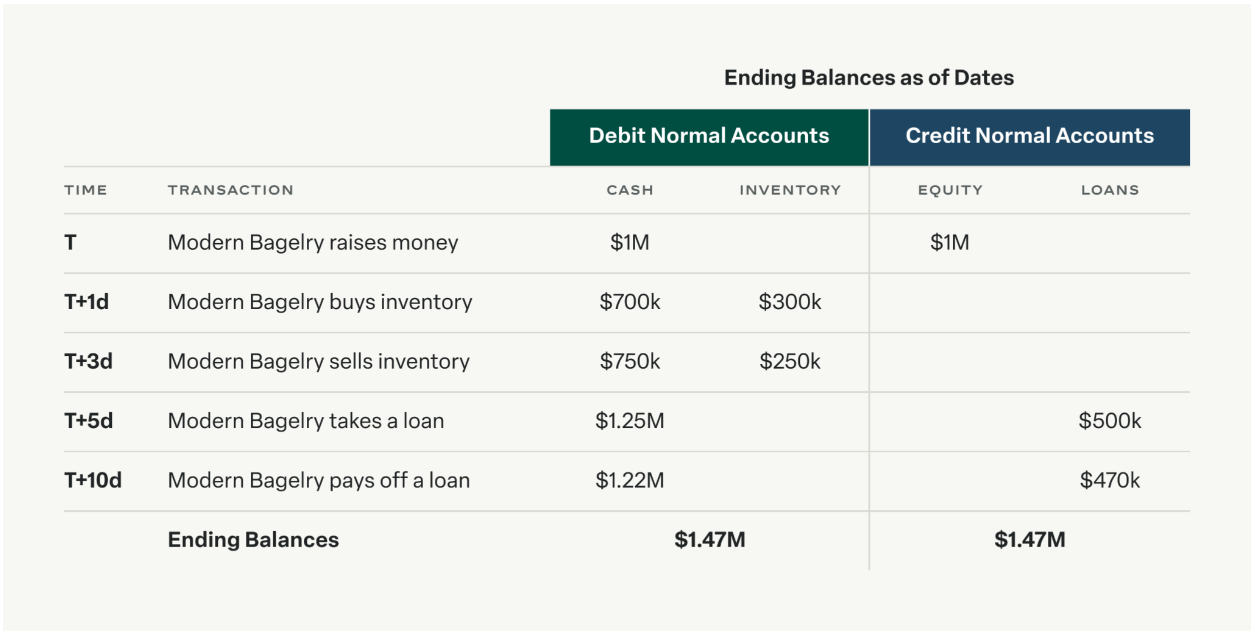
Modern Bagelry example of debits and credits

A common misconception is that one account needs to decrease while another needs to increase. However, they can both increase or decrease in tandem, depending on the debit and credit entries in the transaction and the account types. In the first transaction cash increases because it's a debit entry in a debit normal account (cash); equity also increases because it's a credit entry in a credit normal account (equity). Conversely, in the last transaction both balances decrease because we are adding a debit entry into a credit normal account (loans) and a credit entry into a debit normal account (cash).

Balancing debits and credits

Tracking sources and uses of funds in a single ledger with double-entry is helpful to clearly show clearly that balances match.

Let's say we are aggregating the balances for each account in the example above right after each transaction takes place:



		Ending Balances as of Dates			
		Debit Normal Accounts		Credit Normal Accounts	
TIME	TRANSACTION	CASH	INVENTORY	EQUITY	LOANS
T	Modern Bagelry raises money	$1M		$1M	
T+1d	Modern Bagelry buys inventory	$700k	$300k		
T+3d	Modern Bagelry sells inventory	$750k	$250k		
T+5d	Modern Bagelry takes a loan	$1.25M			$500k
T+10d	Modern Bagelry pays off a loan	$1.22M			$470k
	Ending Balances	**$1.47M**		**$1.47M**	

Ending balances for Modern Bagelry

A system of accounts will balance as long as the balance on debit normal accounts equals the balance on credit normal accounts. The ending balances of **Cash** ($1.22M) and **Inventory** ($250k) sum to $1.47M. That is **equal** to the sum of ending balances of **Equity** ($1M) and **Loans** ($470k). It is said that our accounts in this example are balanced. Not matching would mean the system created or lost money out of nothing.

Before moving on, let's recap the principles we've reviewed so far:

- A ledger is a timestamped log of events that have a monetary impact.
- An account is a discrete pool of value that represents a balance you want to track.
- A transaction is an event recorded in the ledger.
- Transactions must have two or more entries.
- Entries belong to a ledger transaction and also belong to an account.
- Accounts can be classified as credit normal or debit normal.
- Entries can be added onto the ledger "on the debit side" or "on the credit side." Debits and credits refer to how a given entry will affect an account's balance:
- Debits—or entries on the debit side—increase the balance of debit normal accounts, while credits decrease it.
- Credits—or entries on the credit side—increase the balance of credit normal accounts, while debits decrease it.
- If the sum of balances of all credit normal accounts matches the sum of balances of all debit normal accounts in a single ledger, it is said that the ledger is balanced. This is an assurance of consistency and that all money is properly accounted for.

CHAPTER 3

Putting Principles Into Practice: Building A Venmo Clone

In this chapter, we will be designing the ledger for a Venmo clone—a simple digital wallet app. Throughout, we will show how to apply the double-entry accounting principles we covered in the previous two chapters.

If you're curious about the API calls and system design considerations of designing a digital wallet app, you can also check out our previous journal on how to build a digital wallet.

To recap, for a system to gain the consistency benefits that accounting provides:

- It should be composed of accounts and transactions
- Accounts should be classified as debit or credit normal
- Transactions should enforce double-entry upon creation. Each transaction needs to have at least two entries, which, in aggregate, must affect credit and debit sides in equal amounts.
- The aggregate balance of credit normal accounts and debit normal accounts should net out to zero (credits = debits).

Step 1: Reviewing the use case

Let's start with the product requirements of our Venmo clone, first from the user's perspective:

- Each user will have an account balance, exposed via the app
- Users can add to their balance by way of card payments
- Users can send money to each other in the app
- Users can withdraw balances into a bank account via ACH or a debit card

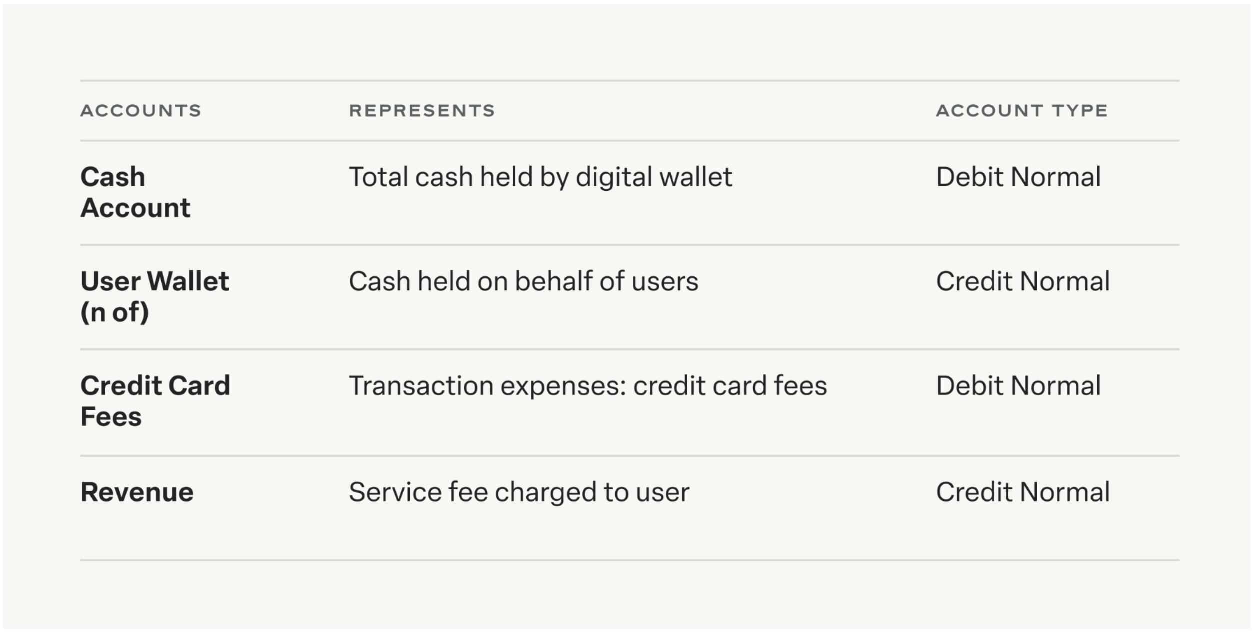
- Users will pay a small fee when they make a withdrawal from the app, to be deducted from their wallet balance.

From a product perspective:

- We want to discern between the account balances for each user and expose them to said users consistently;
- We want to ensure cash in hand in our bank account is always equal to the total users deposited in the app;
- We want to properly calculate and collect revenue from fees;
- Each deposit will need to account for a 3% card transaction processing fee paid by us.

Step 2: Building your chart of accounts

With these requirements in mind, let's map our chart of accounts (COA). The COA is a simple depiction of the accounts we will need, their type, and normality:



ACCOUNTS	REPRESENTS	ACCOUNT TYPE
Cash Account	Total cash held by digital wallet	Debit Normal
User Wallet (n of)	Cash held on behalf of users	Credit Normal
Credit Card Fees	Transaction expenses: credit card fees	Debit Normal
Revenue	Service fee charged to user	Credit Normal

Our Venmo Clone's Sample Chart of Accounts

Let's review this in detail:

A **cash account** represents the amount of money we are holding in our bank account in cash. Because it represents an **asset** or **use** of funds, we will treat it as a **debit normal** account. For more information on debit and credit normal accounts, refer to the "Dual Aspect" section of Part I of this series here.

The user balance accounts represent funds we are holding on behalf of our users. Because users should be able to withdraw them at any time, they are funds we 'owe'—or **liabilities**. Those funds are technically now available for our 'use' - and as such, they are **sources** of funds. Therefore, they should be **credit normal** accounts. Notice that we need one account for each customer that creates an account with us.

To track card fees, we will be using a debit normal account. This account's balance will increase every time we pay off card processing fees. This is a debit normal account because it represents **expenses** or **uses** of funds.

Finally, the fees we collect in each transaction are to be treated as revenue. Given these are **sources of funds**, they are credit normal accounts.

Step 3: Mapping sample transactions

After mapping our chart of accounts, we should consider the typical events that will affect the ledger. For the sake of this example, we will cover three transaction types:

❶ **Transfers**

The user sends money from their balance to another user.

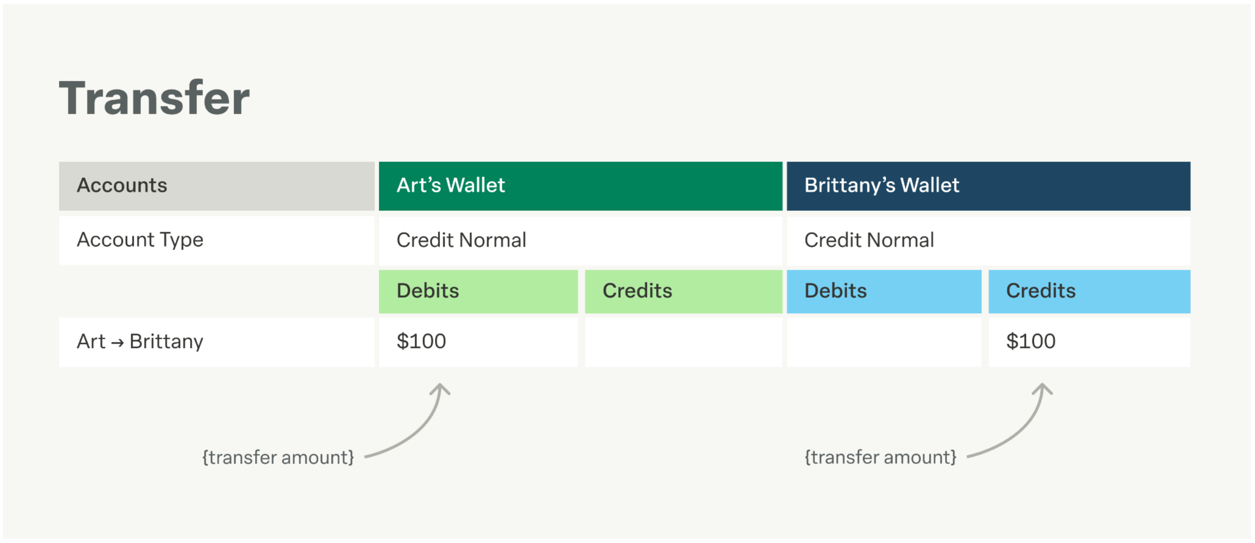
2. **Deposits**

   The user adds cash into their account balance. At the time of transfer, we need to account for the credit card processing fee. (Let's assume, for the sake of this example, that credit card fees are paid by us.)

3. **Withdrawals**

   The user withdraws from their account balance. We charge a fee when users withdraw from the app, deducted from their balance. At the time of transfer, we need to account for our own service fee as revenue.

Let's walk through the implementation for these transactions, starting with a transfer:



Transfer

Accounts	Art's Wallet		Brittany's Wallet	
Account Type	Credit Normal		Credit Normal	
	Debits	Credits	Debits	Credits
Art → Brittany	$100			$100

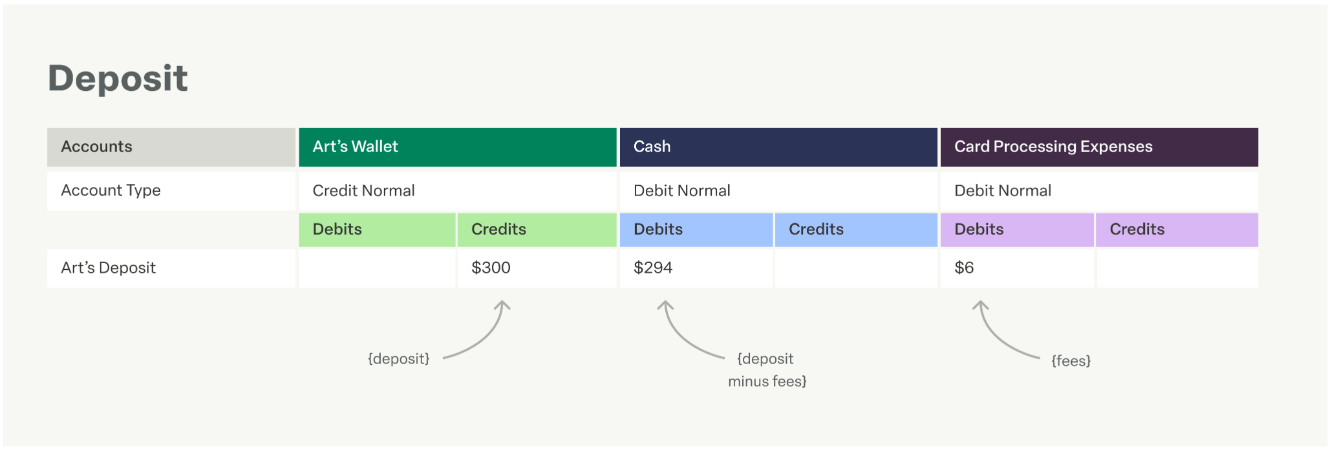
{transfer amount}
{transfer amount}

Mapped funds transfer of $100 from Art to Brittany

This chart shows a typical transfer of $100 from Art to Brittany. In this case, the transaction amount is debited (deducted) from Art's Wallet (who's initiating the transfer) and credited (added) to Brittany's Wallet (who's the receiver).

Note that this logic can be used for any in-app transfer—we just have to designate which wallet is the sender vs receiver in each case. All wallets are represented as credit normal accounts. If Brittany was sending money to Art, then Brittany’s balance would be debited (decrease), and Art’s balance would be credited (increase).

Next, let's look at a deposit:



Deposit

Accounts	Art’s Wallet		Cash		Card Processing Expenses	
Account Type	Credit Normal		Debit Normal		Debit Normal	
	Debits	Credits	Debits	Credits	Debits	Credits
Art’s Deposit		$300	$294		$6	

{deposit}
{deposit minus fees}
{fees}

Art's Deposit of $300

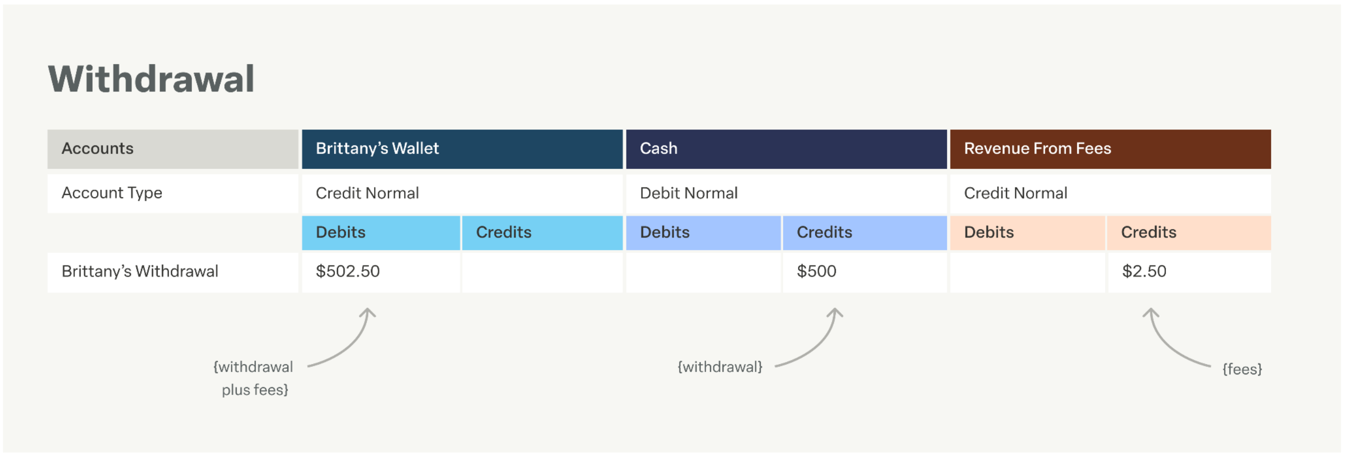
In this model, three accounts are involved: the Art's Wallet, Cash, and Card Processing Expenses. When Art deposits an amount into his wallet, he will see the balance increase by the same amount. Simultaneously this will increase cash balance and the total paid in processing fees.

To further illustrate this, let’s say Art deposits $300 in his wallet balance using a credit card. Recall that for the sake of this example, our app is paying for card fees. To counterbalance the $300 credit (increase) on Art’s Wallet, we need two debit entries: one on the cash account (which increases it) and one on the card processing expenses account (which also increases it).

Our card processing expenses account increases by $6 (or 2% of the transaction). And given we are recording this expense as paid off to our credit card vendor, our cash balance increases by $294 ($300-$6).

The power of double-entry is recording this flow of money in a single event. Without double-entry, we would need a way for the system to recognize all of the deposit transactions and properly account for card fees. By recording all of the money movement in a single transaction with multiple entries, we make sure our system is consistent. As debits = credits, money in equals money out.

The same goes for a withdrawal:



Withdrawal

Accounts	Brittany's Wallet		Cash		Revenue From Fees	
Account Type	Credit Normal		Debit Normal		Credit Normal	
	Debits	Credits	Debits	Credits	Debits	Credits
Brittany's Withdrawal	$502.50			$500		$2.50

{withdrawal plus fees}
{withdrawal}
{fees}

Brittany's withdrawal of $500

A withdrawal is similar to a deposit, except that in this case, we are charging an extra fee from the user and recognizing it as revenue from fees. This transaction will decrease Brittany's Wallet and Cash but will increase Revenue from Fees.

For example, let's say Brittany is withdrawing $500 from her wallet balance. Brittany knows that she will pay a fee on that transfer amount. Let's assume that the fee is 0.5% of the withdrawal amount, or $2.50. Her user wallet gets deducted for the entire $500 + $2.50, or $502.50. That is the debit entry (decrease) on her user wallet balance.

To represent this on the credits side, we will add a credit entry that deducts the cash account for $500, given this is actual money we wired out to Brittany. However, we owe $2.50 less to Brittany and can recognize the fees we charged from her as revenue by crediting (increasing) our revenue from fees account.

There are many different ways to model this. We could have chosen to have Brittany receive $497.50 ($500-$2.50), for example. In this case, we would add/credit the $2.50 we kept to revenue from fees similarly, but our cash would only decrease/credit by $497.50. The ledger would still balance. Thinking in terms of credit and debit normality gives you the flexibility to log transactions in the best way for your business.

Step 4: Bringing it all together

Let's review the logical elements we would create to service this use case:

- One ledger object that represents the entire collection of accounts and transactions. All of our accounts and transactions should belong to a single ledger.
- At least four types of account objects:
    - User Wallets (one per user, credit normal)
    - Cash (single account, debit normal)
    - Revenue from Fees (single account, credit normal)
    - Card Processing Expenses (single account, debit normal)
- At least three modeled transactions
    - User Transfer
    - Deposit
    - Withdrawal

If you are building a ledger using a relational database you'll want to model accounts as belonging to a single ledger where transactions (or events) will be written into. Accounts should have constraints according to their type: credit or debit normal. Such constraints should dictate how debit or credit entries affect account balances according to the principles we covered in our first two chapters.

Similarly, transactions would have to be modeled in a way such that they are composed of at least two entries. Such entries would have a 'direction', one of debit or credit. Your system should enforce equality between the sum of amounts on debit entries and the sum of amounts on credit entries.

The double-entry treatment for each type of transaction we covered on step 3 can then be mapped into functions in your application code that dictate how you will write into the ledger database as transactions happen.

By setting up the ledger as a double-entry system, we ensure that our Venmo clone scales consistently. And as new product requirements come up or functionalities are rolled out, we can update our chart of accounts and the transaction models to represent them in the ledger appropriately.

CHAPTER 4

Applying The Concepts: Building a Lending Marketplace

This chapter will explore the accounting principles behind a lending app. Here, we'll look at a consumer lending marketplace akin to Lending Club that we'll call Modern Lending. While we are showcasing a lending marketplace, you will find that the principles presented here are applicable to most types of lending ledgers.

For lending applications, keeping track of money using double-entry is especially important. Double-entry ensures funds are not created or destroyed out of nothing, and balances are always accurate. Using double-entry constraints in database development is a best practice for fintechs and other companies that move money (read more about this topic in our article How And Why Homegrown Ledgers Break and the public examples of Uber, Square, and Airbnb).

Why Double Entry

As a reminder, obeying double-entry accounting rules boils down to following these principles:

- Your ledger should be composed of accounts and transactions;
- Accounts represent the balances your ledger will track. They can be classified as debit normal or credit normal.
- Transactions represent business events that have a monetary impact. They are composed of multiple entries (at least two). Each entry can be on the debit side or credit side.
- The aggregate balance of all credit normal accounts and all debit normal accounts in a ledger should net out to zero (credits = debits).

Step 1: Reviewing The Use Case

Let's start with the requirements for our fictitious Modern Lending. This is a peer-to-peer lending marketplace where individuals can add money to a shared pool of cash managed by Modern Lending. These investors can then set their risk preferences, accept interest rates and invest directly on the platform.

Modern Lending then lends out the money to borrowers on the platform, adjusting the amount and interest rate based on their creditworthiness. Borrowers pay back their principal balance over time, and after repayments, Modern Lending returns capital to investors, plus interest. Modern Lending recognizes (books) the spread between the interest it gets paid by borrowers and the interest it pays to investors as revenue.

This translates into the following set of product requirements:

- Users should be able to self-select as investors or borrowers during onboarding;
- **For investors:**

    - During onboarding, they would choose between a set of investing options, varying by term (when they get their money back) and interest rate (how much they get in return).
    - After onboarding, they would wire or ACH their committed investment into the Modern Lending cash pool. This investment would be represented in their account as a balance.
    - At the end of their chosen term, they receive their principal back plus interest.
- **For borrowers:**
    - During onboarding, they would submit information about creditworthiness and income.
    - They would then receive a list of potential loans they can take varying by amount, term, and interest rate.
    - Next, they would pick one of these loans and have money disbursed to them in the form of a wire, ACH, or RTP transfer.
    - Over time, the borrower would make repayments in the platform against their principal and interest balance, which would be represented in-app.

Let's assume there are no transaction costs or fees. We will also assume we are building this application for scale—Modern Lending should be configured to handle thousands of investors and borrowers triggering disbursements and collections every day. Finally, let's also assume we would like to build our ledger flexibly to accommodate future product expansions.

This scenario creates a set of **ledgering requirements**:

❶ Transactions need to be logged as they happen.

For this, we will need a ledger API that embeds directly into our application code and writes into the ledger database as business events happen. Modern Lending needs to parse financial transactions and dictate how to write in the ledger—we'll call this 'translation' service *transaction handling logic*.

❷ **Balances need to be kept up to date consistently and automatically.**

Transactions need to be parsed appropriately in order to update the correct balances. Aggregations need to be efficient, and balances need to be updated within milliseconds of a transaction taking place. They also need to be queryable to support transactions such as showing the user an updated balance after a transaction is completed. For this to be true, we need to map our accounts to their given normality—a set of constraints that will help the ledger obey double-entry rules. Such constraints are represented in a *chart of accounts*.

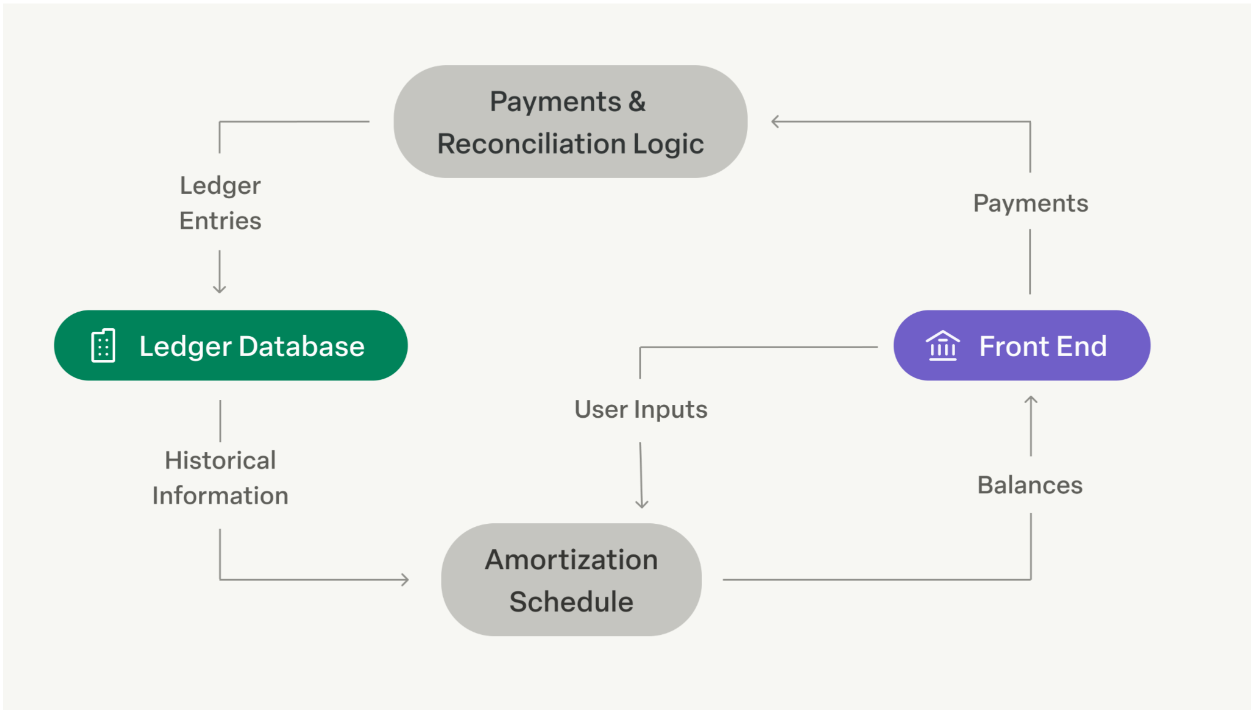
❸ **The ledger needs to accommodate a high volume of financial transactions and be built flexibly to expand our product offerings in the future.**

In essence, this needs to be a central ledger that is divorced from fund movements (i.e., our underlying bank account setup) and is fully programmatic and flexible. The ledger's functionality and underlying data models should not be tightly coupled to business logic.

In the next sections, we discuss the accounting setup—first, our *chart of accounts*, then our *transaction handling logic*.

Step 2: Reviewing The Data Flow

Before we start with the ledgering setup, it's important for us to understand how data flows in a lending system. Modern Lending would need to implement the following services infrastructure alongside its ledger.



How Modern Lending's systems come together

There are two kinds of data Modern Lending needs to keep track of:

- **Historical data**, or data that reliably represents the current financial state of Modern Lending based on posted transactions.
- **Prospective data**, specifically those that are defined by the business model: interest rates, loan terms, payment amounts, and the payment breakdown of interest versus principal.

Modern Lending will need to make use of an amortization schedule, a tool that utilizes interest rates, principal, and loan term to define monthly payments. It is also common for lending businesses to front-load the percentage of a customer's payments that goes towards interest versus principal. An internal service on Modern Lending's backend can track this data and modify outputs based on new inputs (i.e., new loan terms).

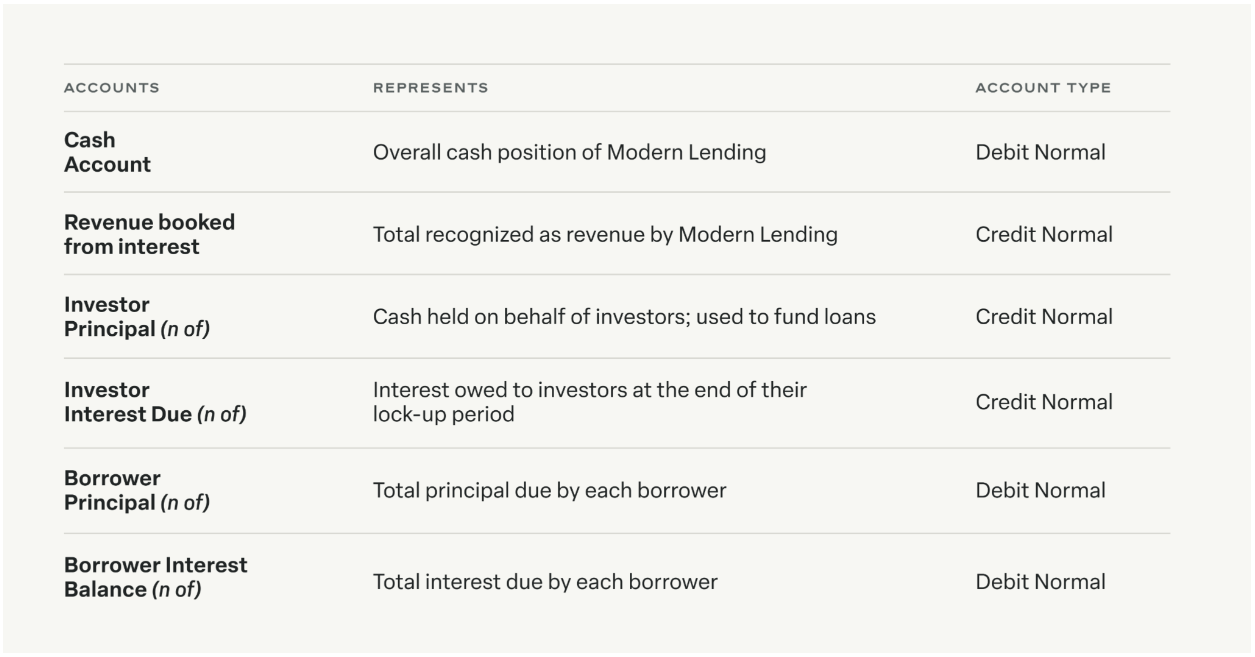
The ledger database acts in tandem with the amortization schedule. The ledger acts as the source of truth for historical information. Every time the amortization schedule service needs to provide an updated view of current balances, it cross-references the amortization schedule with the ledger database to provide accurate data.

Keeping historical information in the ledger and the amortization schedule outside of the ledger keeps Modern Lending's data store clean and referenceable over the course of the lifetime of the loan.

Notice that the ledger also needs to interact with another service divorced from the underlying ledger: payments and reconciliation logic. This service is our way of representing Modern Lending's payment processor of choice: this can be a card processor or Modern Treasury Payments API if you're using ACH, wire, or RTP. Transactions come in as webhooks and get parsed as ledger transactions according to the rules presented below.

Step 3: Building Your Chart of Accounts

A [chart of accounts]() is a simple depiction of the accounts Modern Lending will need, their type, and normality.



ACCOUNTS	REPRESENTS	ACCOUNT TYPE
Cash Account	Overall cash position of Modern Lending	Debit Normal
Revenue booked from interest	Total recognized as revenue by Modern Lending	Credit Normal
Investor Principal ***(n of)***	Cash held on behalf of investors; used to fund loans	Credit Normal
Investor Interest Due ***(n of)***	Interest owed to investors at the end of their lock-up period	Credit Normal
Borrower Principal ***(n of)***	Total principal due by each borrower	Debit Normal
Borrower Interest Balance ***(n of)***	Total interest due by each borrower	Debit Normal

Modern Lending's Chart of Accounts

Let's review this in detail. First, we have two general accounts: cash and revenue. **Cash** tracks the overall cash position of Modern Lending. As it represents a **use of funds**, it is a **debit normal** account. Revenue, on the other hand, represents a tally of how much money we recognize (book) as revenue during the regular course of business. It is a **credit normal** account because it represents a **source** of funds. For the sake of this example, Modern Lending has a single revenue stream: interest.

You may have noticed we have a set of "n of" accounts. This is our way of representing that **each** investor and **each** borrower will have two sets of accounts: one tracking principal and one tracking interest. While we will only need one **cash** and one **revenue** account, we need multiple sets of user accounts.

Principal accounts track total capital invested by investors and total capital lent to borrowers. The investor principal accounts are **credit normal** because they represent **sources** of funds—or funds Modern Lending owes. Conversely, borrower principal accounts are **debit normal** because they represent **uses of funds**—akin to receivables.

Finally, **interest accounts** track the interest balance for both investors and borrowers. They follow the same normality rules as the principal accounts: they are **credit normal** when they track interest due to investors and **debit normal** when they track interest that is owed to Modern Lending by the borrowers.

Step 4: Mapping Transaction Logic

Let's consider the typical set of events that will hit the ledger. As mentioned before, a stream of transactions is parsed through the internal service handling payments and reconciliation. Transactions are then written into the ledger according to the rules presented below. The examples below are meant to be illustrative of what transaction handling logic looks like - there are other transaction types not presented here Modern Lending would need to parse out.

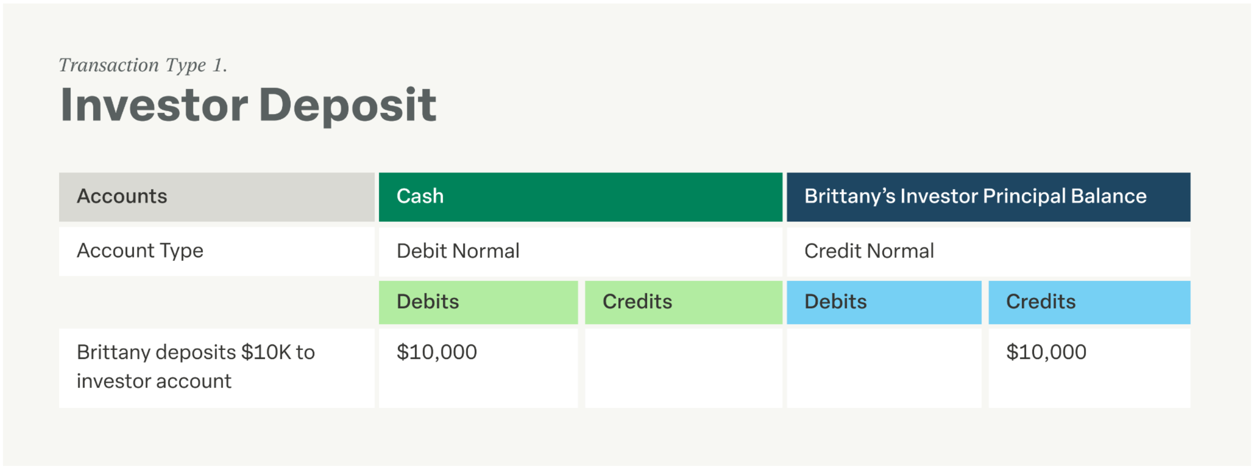
Here we cover the following archetypical transactions:

- **Investor deposit.** An investor adds money to their balance.
- **Borrower disbursement.** A borrower initiates a loan and receives funds.
- **Interest calculation.** Monthly interest gets added to the interest balance for both borrowers and investors. As part of this calculation, part of the interest gets recognized as booked revenue.
- **Borrower repayment.** A borrower repays monthly installments covering both interest and principal.
- **Investor withdrawal.** At the end of their term, Modern Lending sends principal plus interest owed to investors.

In this example, we'll simplify our marketplace down to one investor (Brittany) and one borrower (Art). We'll also assume the following data about our loans is coming from our amortization schedule:

- Brittany is depositing $10,000 as an investor and expects a 4.8% return upon completion of her 1-year term. This equates to a payment of $10,480 to her at the end of month 12, or $10,000 in principal plus $480 in interest.
- Art is borrowing $5,000 and will pay 12% annual interest over the course of his 1-year loan. This equates to $600 in interest at the end of the year. If we also assume simple interest calculations and monthly payments, the amortization schedule also informs our ledger that Art needs to make monthly payments of $466.67, with $416.67 ($5,000 / 12) of this being directed towards principal, and $50 ($600 / 12) directed towards interest.

Transaction Type 1. Investor deposit



Transaction Type 1.

Investor Deposit

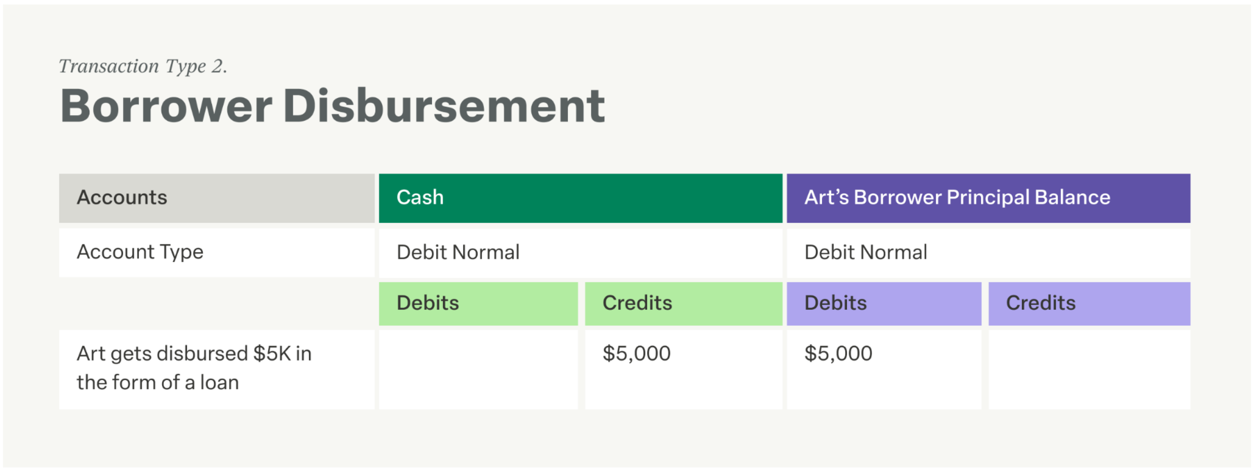
Accounts	Cash		Brittany’s Investor Principal Balance	
Account Type	Debit Normal		Credit Normal	
	Debits	Credits	Debits	Credits
Brittany deposits $10K to investor account	$10,000			$10,000

Brittany's $10K investor deposit

As Brittany deposits money on the Modern Lending platform, we **debit (increase)** the cash account and **credit (increase)** Brittany’s investor balance account. Brittany’s interest rate is fetched from our amortization schedule.

Notice that we don’t record any kind of interest due to Brittany on the ledger at this time, regardless of the fact Brittany was promised a 4.8% return upon sign-up. Interest is ledgered in a separate transaction (see below).

Transaction Type 2. Borrower disbursement



Transaction Type 2.

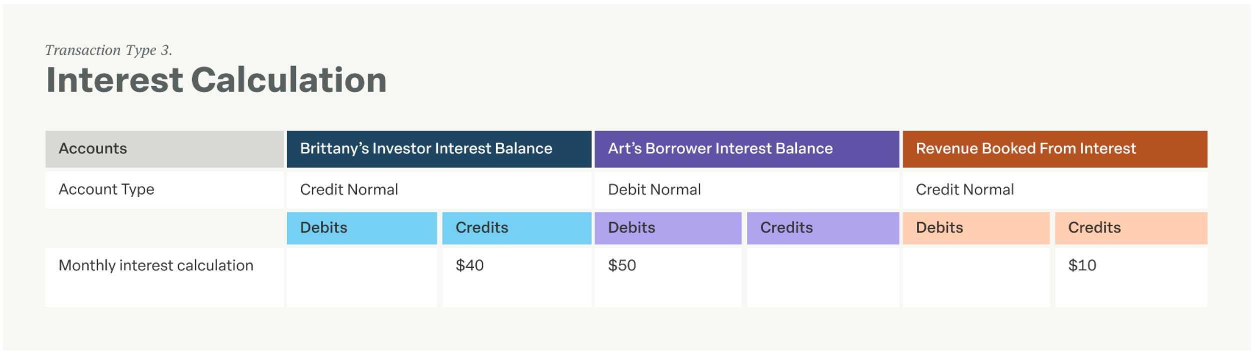
Borrower Disbursement

Accounts	Cash		Art’s Borrower Principal Balance	
Account Type	Debit Normal		Debit Normal	
	Debits	Credits	Debits	Credits
Art gets disbursed $5K in the form of a loan		$5,000	$5,000	

Art gets disbursed $5K

When Art is approved for a loan and receives $5,000, we **credit (decrease)** our cash account and **debit (increase)** Art's principal due account. As above, notice we do not record any kind of interest owed by Art at this time.

Transaction Type 3. Interest calculation



Transaction Type 3.

Interest Calculation

Accounts	Brittany's Investor Interest Balance		Art's Borrower Interest Balance		Revenue Booked From Interest	
Account Type	Credit Normal		Debit Normal		Credit Normal	
	Debits	Credits	Debits	Credits	Debits	Credits
Monthly interest calculation		$40	$50			$10

Monthly interest calculation

Recall that Brittany lent $10,000 and expects a 4.8% return in a year. That means her effective interest payment at maturity (when the loan is due) equates to $480. Given we are assuming simple interest, we can see that we should add $40 to her interest balance to the ledger every month ($480/12). This is a **credit** because we are increasing a **credit normal** account.

As outlined before, Art borrowed $5,000 at 12% to be paid in a year. This means Art's final interest balance will be $600 at the end of the year. Assuming simple interest again, our ledger should add $50 every month to Art's interest balance ($600/ 12). This is a **debit** because we are increasing a **debit normal** account. (Let's assume Brittany's remaining $5,000 just sits on Modern Lending's cash account for now).

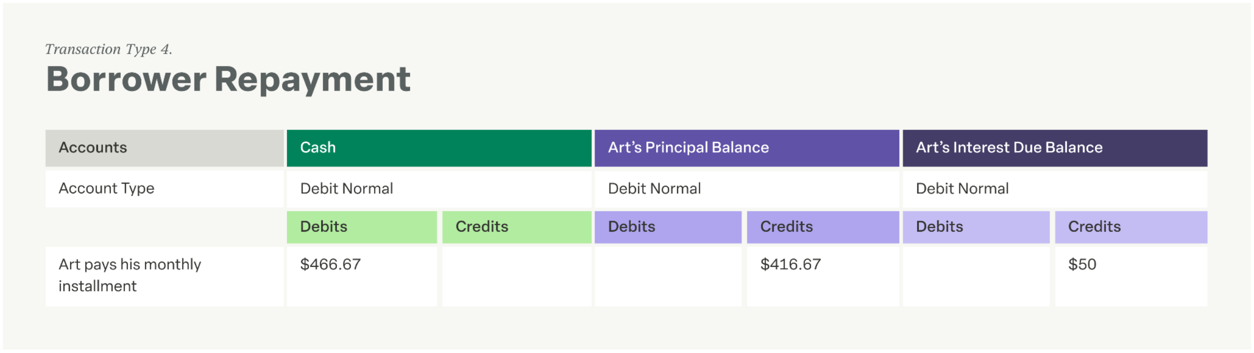
At the end of the year, Modern Lending will have received $600 from Art and will owe Brittany $480 in interest. The difference of $120 is recognized as revenue. Every month, as interest gets calculated, we add $10 to our revenue account ($120 / 12).

You may be wondering why we recognize revenue as interest gets calculated, as opposed to when a borrower pays out their repayments or when an investor receives their payout. There is a difference between booked revenue (accrual) and realized revenue (cash). These two concepts are borne from accrual and cash accounting principles.

In this case, we are using accrual accounting and booking revenue at the point in which it is earned: when interest is incurred. In a marketplace of hundreds or thousands, these transactions would be handled by a monthly cron job that calculates interest and modifies balances on the ledger at a predetermined date.

Were we using cash accounting, we would add to the realized ledger only after a borrower makes their repayments and investors get paid back. Because your application ledger doesn't need to comply with GAAP accounting rules, we suggest you pick the method that leads to the cleanest ledger.

Transaction Type 4. Borrower repayment



Transaction Type 4.

Borrower Repayment

Accounts	Cash		Art's Principal Balance		Art's Interest Due Balance	
Account Type	Debit Normal		Debit Normal		Debit Normal	
	Debits	Credits	Debits	Credits	Debits	Credits
Art pays his monthly installment	$466.67			$416.67		$50

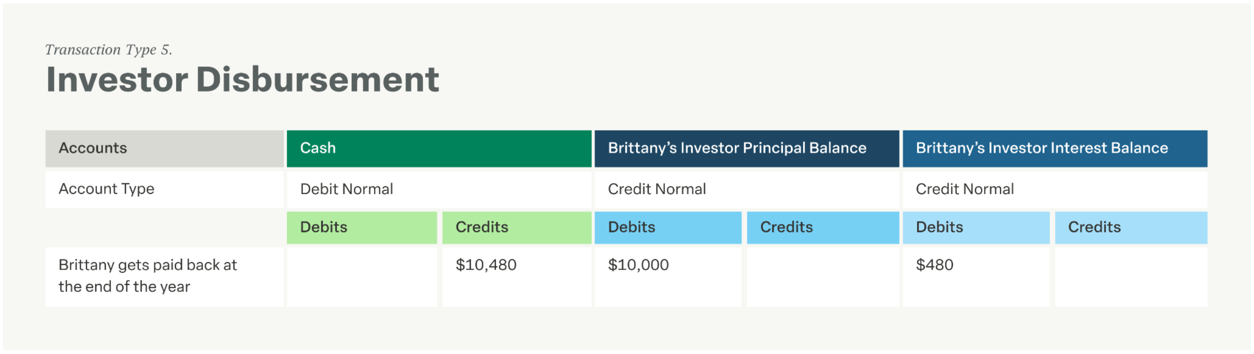
Art pays his monthly installment

Every month Art makes a payment towards his balance. At the end of the term of his loan Art will have to pay a total of $5,600, $5,000 in principal, and $600 in interest. This translates into a monthly payment of $466.67.

After the transaction clears, we **debit (increase)** our cash account by $466.67. In the same transaction, we **credit (decrease)** both principal and interest balances for Art. Principal gets deducted by $416.67 ($5,000 / 12) and interest gets deducted by $50 ($600 / 12).

At this point, the ledger shows one payment made by Art, a principal balance of $4,583.33 ($5,000 - $416.67), and an interest balance of zero. This happens because we recognized interest in the "interest calculation" section and then immediately zeroed it out as the payment was made. The sum of credits towards Art's interest balance represents the total paid in interest. We can contrast this with the data from Modern Lending's amortization schedule to derive Art's remaining principal balance.

Transaction Type 5. Investor disbursement



Transaction Type 5.

Investor Disbursement

Accounts	Cash		Brittany's Investor Principal Balance		Brittany's Investor Interest Balance	
Account Type	Debit Normal		Credit Normal		Credit Normal	
	Debits	Credits	Debits	Credits	Debits	Credits
Brittany gets paid back at the end of the year		$10,480	$10,000		$480	

Brittany gets repaid at year end

Finally, at the end of the year, Brittany would get disbursed her principal of $10,000 in addition to the 4.8% return—or $480—promised to her. We **credit (decrease)** our cash account, and simultaneously, we lower her principal and interest down to zero by **debiting (decreasing)** $10,000 and $480, respectively.

Step 5: Bringing It All Together

Let's review the elements we would need in place to service this use case:

- Modern Lending **needs a central ledger** that represents the entire collection of accounts and transactions;

- Working with the ledger, **we need two services**: one that streams transaction data (typically a payments API like Modern Treasury) and one amortization schedule.
- **The ledger will need *at least* six types of accounts:**
    - One debit normal cash account
    - One credit normal revenue from interest account
    - One credit normal investor principal account per investor in the platform
    - One credit normal investor interest account per investor in the platform
    - One debit normal borrower interest account per borrower in the platform
    - One debit normal borrower principal account per borrower in the platform.
- **Transaction logic that supports *at least the following five types of typical transactions*:**
    - Investor deposit
    - Borrower disbursement
    - Interest calculation
    - Borrower repayment
    - Investor disbursement

The bottom line is that while using double-entry is the best way to ensure integrity of the financial information flowing through Modern Lending's product, it requires a bit of setup and parsing through the most common transactions. While this seems hard at first, understanding accounting principles goes a long way. The hard part—we believe—is setting up the underlying ledger database with the right constraints and level of flexibility.

In Summary

It can be onerous for generic databases to reliably handle double-entry accounting. If you are a developer who works with money, the opportunity cost of building a ledger from scratch may not be worth it. Modern Treasury Ledgers simplifies the process of building a dependable double-entry system. Reach out to us to learn more.