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

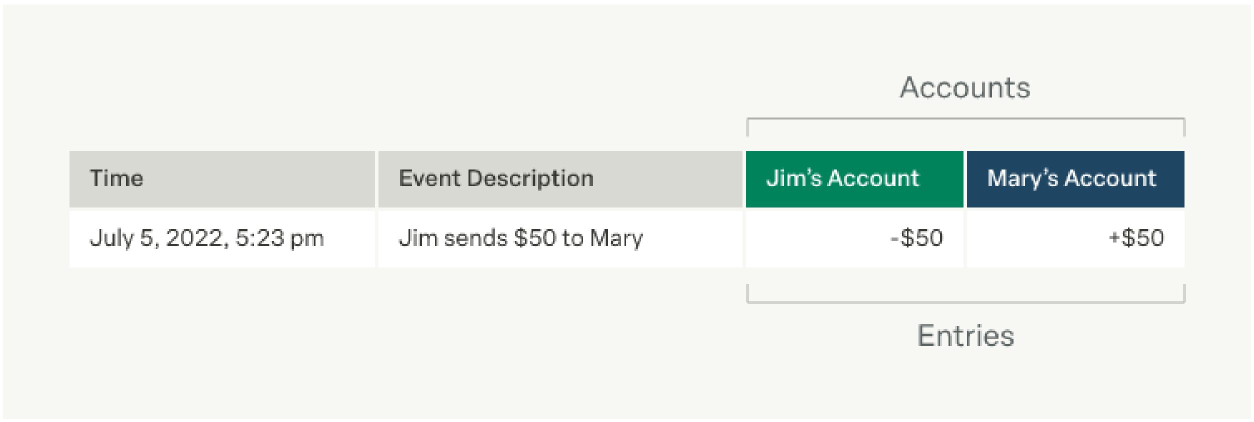
An account is a segregated pool of value. The easiest analogy here is your own bank checking account: money that a bank is holding on your behalf, clearly demarcated as yours. Any discrete balance can be an account: from a user's balance on Venmo to the annual defense spending of the United States. Accounts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

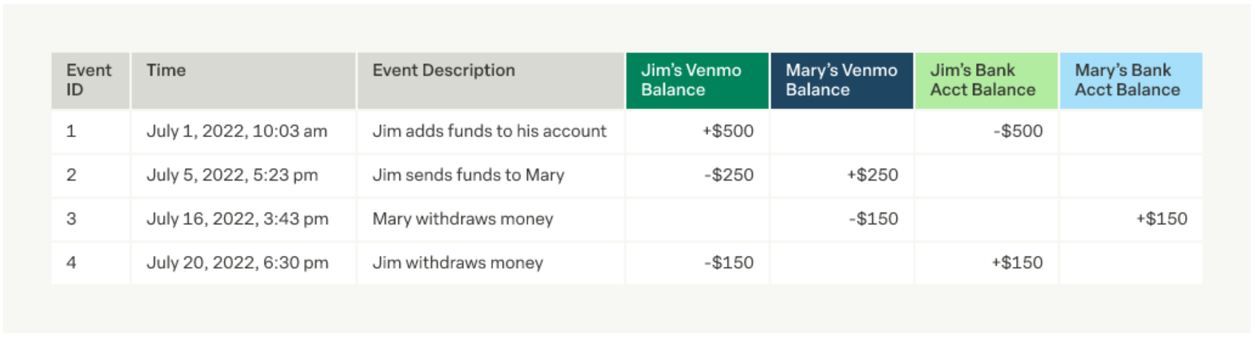
Let's use a simple Venmo transfer as an example. Jim is sending $50 to Mary:


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	

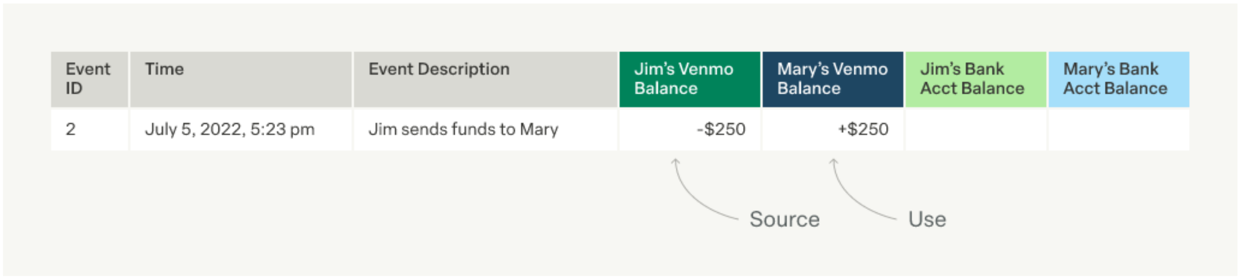
Simple ledger of Jim and Mary's accounts

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

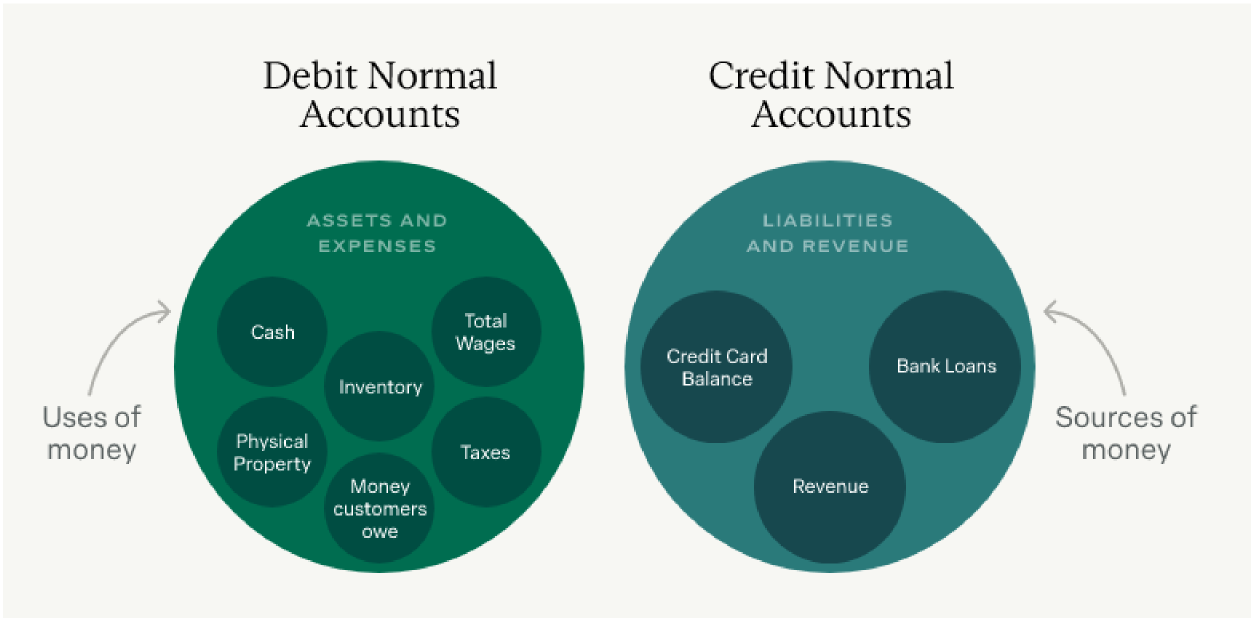
As mentioned, another big innovation of accounting was creating account types. The two types we will cover here are debit normal and credit normal:

- 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.

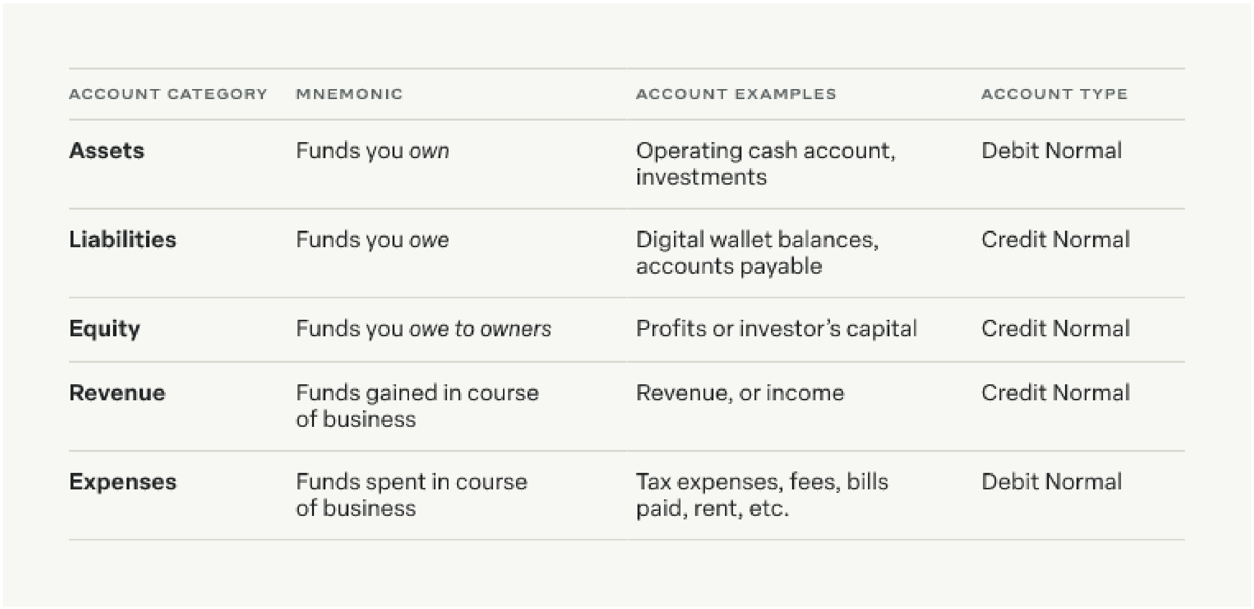
Examples of **uses** of funds are **assets** and **expenses**. Buying inventory, making an investment, acquiring physical property, and so on. The term "use" is broadly defined: letting cash sit in a bank account is a use of funds, as well as selling on credit to someone else (you are effectively "using" 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

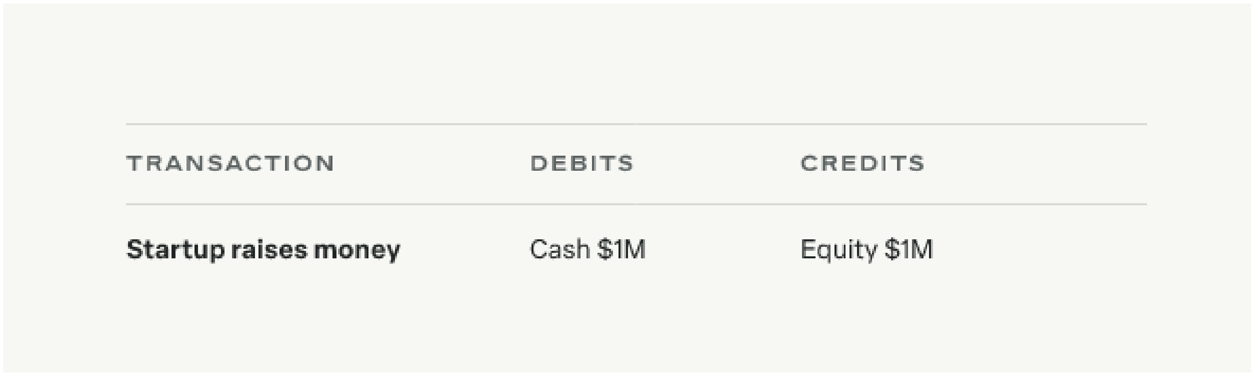
Account categories with simple mnemonics and examples

(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:

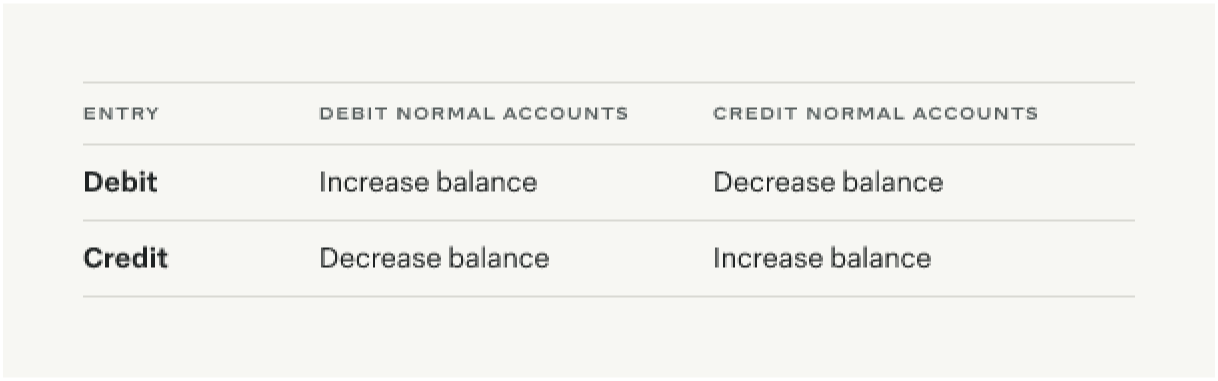


TRANSACTION	DEBITS	CREDITS
Startup raises money	Cash $1M	Equity $1M

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:

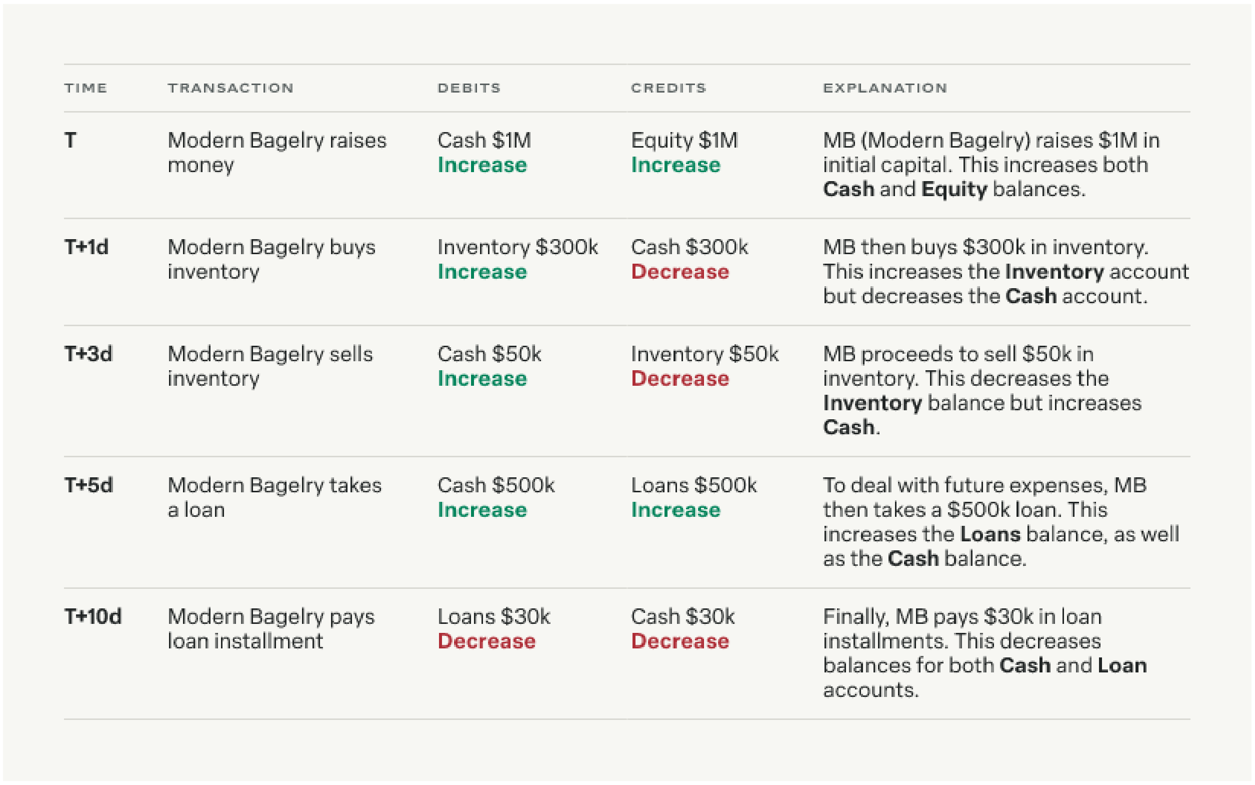


ENTRY	DEBIT NORMAL ACCOUNTS	CREDIT NORMAL ACCOUNTS
Debit	Increase balance	Decrease balance
Credit	Decrease balance	Increase balance

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.

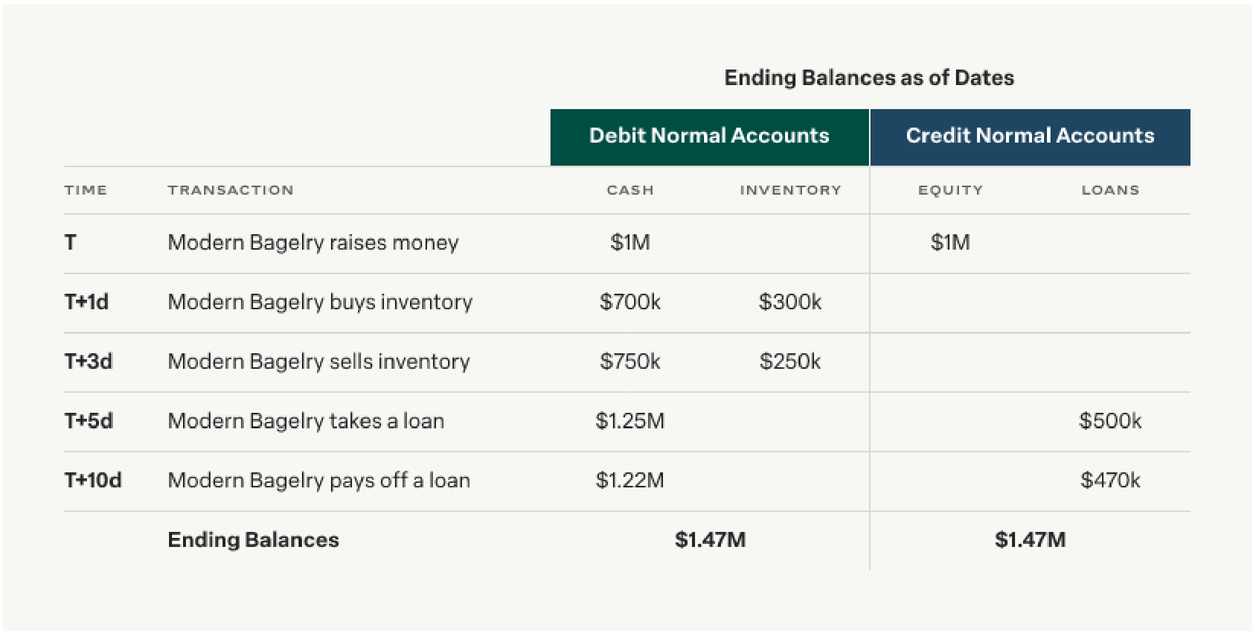
Modern Bagelry example of debits and credits

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

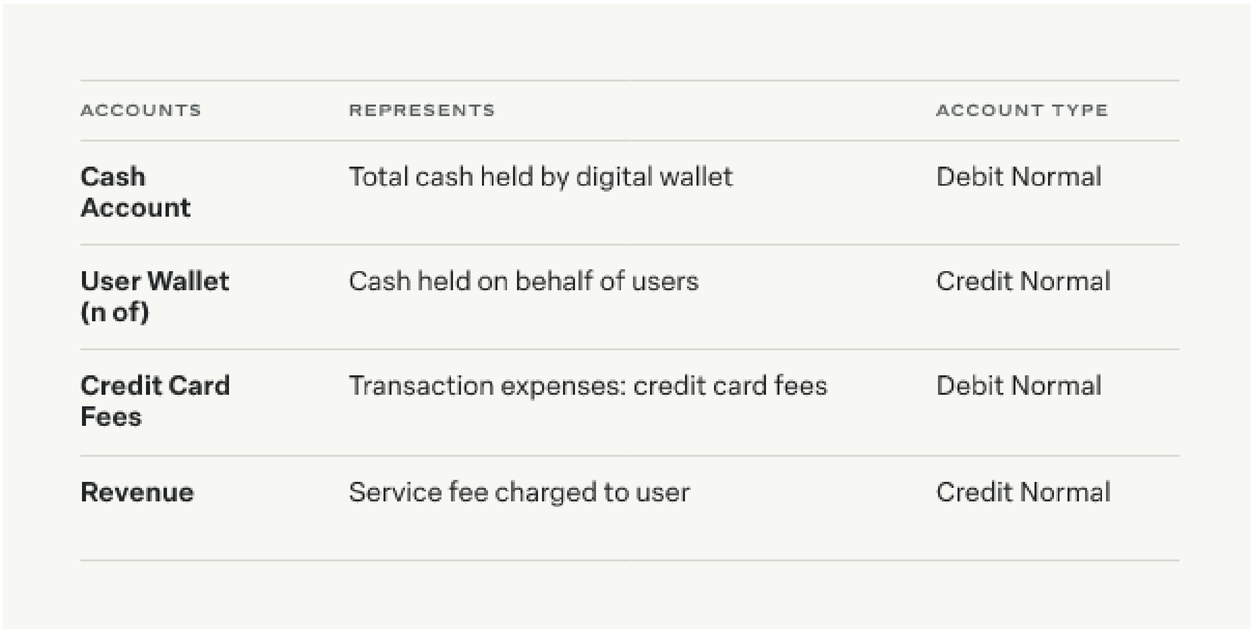
- Users will pay a small fee when they make a withdrawal from the app, to be deducted from their wallet balance.

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:

1. **Transfers**

   The user sends money from their balance to another user.

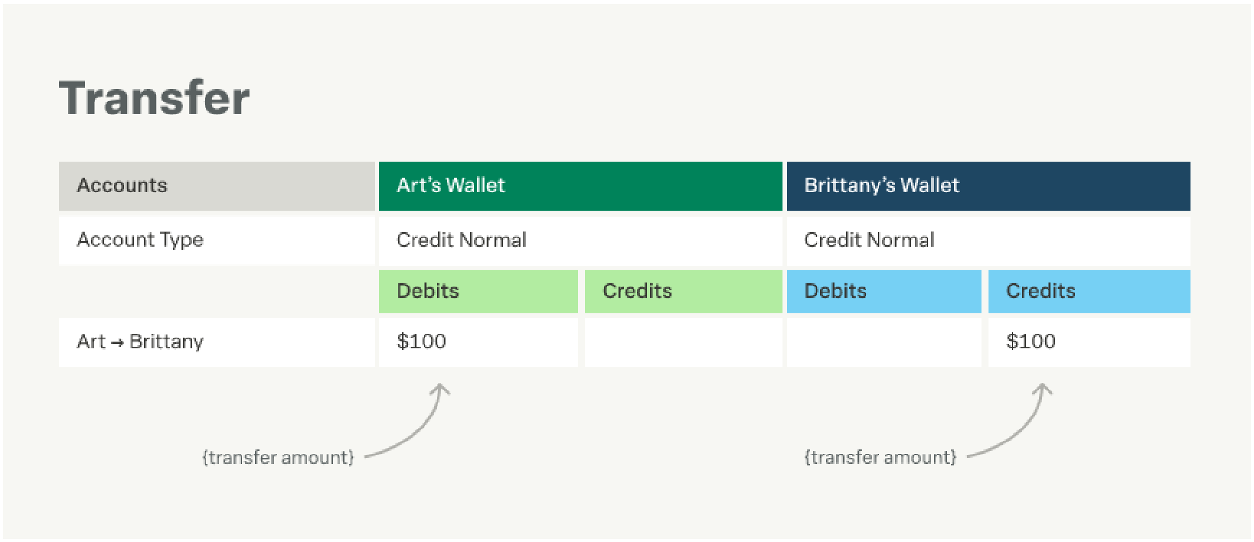
❷ **Deposits**

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.)

❸ **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

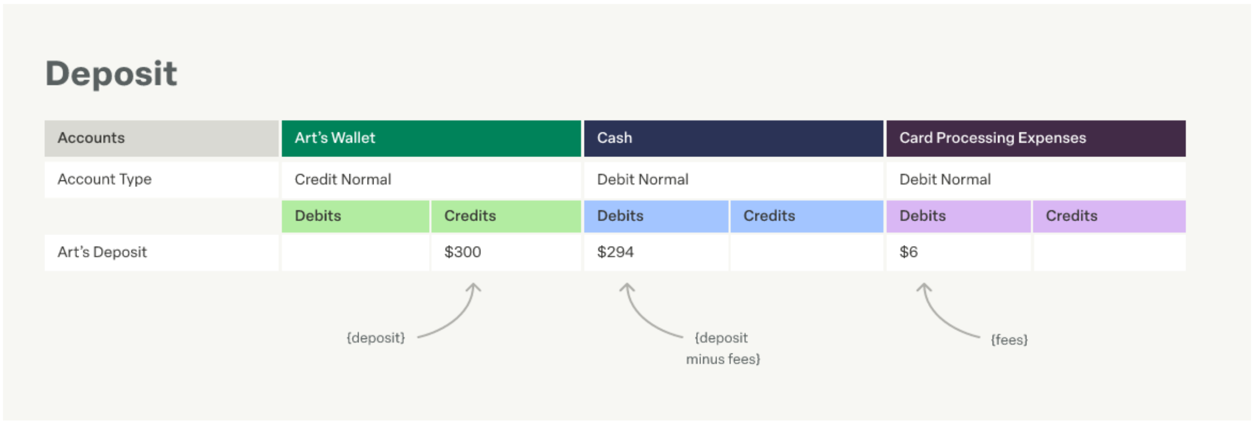
{transfer amount}
{transfer amount}

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

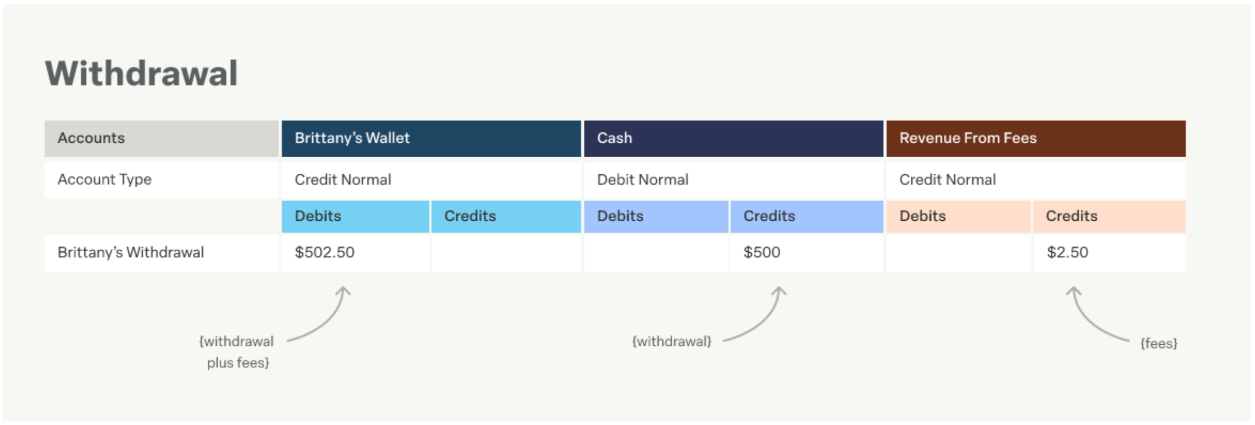
In this model, three accounts are involved: the Art's Wallet, Cash, and Card Processing Expenses. When Art deposits an amount into his wallet, he will see the balance increase by the same amount. Simultaneously this will increase cash balance and the total paid in processing fees.

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.

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.