Familienkonto - Technical Manual

Architecture, data model, authentication, deployment and the command line.

Written by mplx <developer@mplx.eu>

2026-09-06

Contents

Overview

Familienkonto is an expense-and-settlement ledger for a family, with Austrian family-allowance reporting on top. It is written in Jennifer (https://jennifer-lang.dev/) and stores its data in MariaDB.

> **This manual as a PDF (../familienkonto-technical.pdf)** - every technical page in one file. The German handbook for the people who *use* the application is a separate download: **Handbuch (../familienkonto-handbuch.pdf)** .

Shape of the code

```
src/
  core/    decisions and values: money, dates, currency, roles, the ledger,
           coverage, the notification catalogue, the message catalogue. No
           database; testable on a bare machine
  store/   everything that talks to MariaDB: the schema and its migrations,
           people, expenses, income, settlements, attachments, the CSV import
  report/  reads the store and writes documents: reports and the tax bundle
  app/     the layer the entry points stand on: authentication, demo data
  testing/ fixtures, used only by tests
web/     the HTTP interface: routes, handlers, HTML rendering
bin/fk   the command-line tool
```

The dependencies inside `src/` run one way - `core` knows nothing of `store` , `store` knows nothing of `report` or `app` - which is what keeps the pure arithmetic testable without a database and stops a reporting question from reaching into the schema.

```
docker/  the image
docs/    this documentation
```

Everything of consequence lives in `src/` . `web/` and `bin/fk` are two front ends over the same modules, which is why the rules cannot differ between them: a child is refused the review queue in the browser and on the command line by the same predicate.

Layers

Layer	Modules	Depends on a database?
Pure domain	money dates currency ledger coverage roles notify bankcsv	no
Configuration	config	no
Persistence	db	yes
Repositories	users expenses incomes receipts settlements notifications mailqueue audit bankimport	yes
Reporting	reports taxexport	yes
Access	auth	yes
Front ends	web/ bin/fk	yes

The pure layer is where the arithmetic lives, and it is deliberately ignorant of storage. `ledger.j` decides who owes whom, `coverage.j` decides whether the family-allowance threshold is met, and neither can be told otherwise by a query. Their tests run without MariaDB.

Modules worth reading first

- **ledger.j** - the settlement model in about 250 lines of arithmetic. If you read one file, read this one; everything about balances follows from it.
- **coverage.j** - the family-allowance rule, in exact integers.
- **expenses.j** - how a payment becomes shares, and what approval means.
- **auth.j** - who is asking, and why a header is not evidence.

Design decisions that shaped everything else

JavaScript is avoided, not forbidden. The rule is: do it on the server if the server can do it, and reach for a script only where leaving it out costs the user something real - then keep it to the smallest thing that buys that back. Every page is a server-rendered form, the dark mode is a CSS media query and a cookie, the split calculator is arithmetic done on POST. The application therefore works in any browser, has no build step for its front end, and cannot leak a session to a script somebody injected.

Nothing may *depend* on a script. Each one is a convenience laid over a page that already works without it, the server re-derives whatever the script computed, and every check happens where it always did.

What has met that bar so far:

- **the payment total under /settlements** (`SETTLE_SCRIPT` , ~30 lines, inline). The amount is the sum of the ticked open items; ticking lines without seeing the total is a hole where the important number should be. With scripting off the box stays empty and means "exactly what is ticked", which is what the server books either way.
- **the photograph on the last quick-entry screen** (`QUICK_SCRIPT`). It scales the picture to 1600 pixels on the long edge before sending it and puts a spinner in the button. A phone camera produces eight to twelve megapixels over a shop's mobile connection, and the server scales the picture down on arrival anyway - so those bytes buy nothing and cost the minute in which somebody gives up. Every failure mode falls back to uploading the original: no script, no `DataTransfer` , no canvas, an image the browser cannot decode.

What has not, and why the bar is worth having:

- **no progress indicator on the ordinary upload form** , only on the quick entry. An identical file is refused with a reason, and the form states the size limit.
- **no client-side resizing outside the quick entry** , so the scaling happens on the server after the whole file has arrived - see Deployment.
- **the Finanzamt checkbox cannot follow the category** as it changes, so the form offers three states and resolves the default on submit.

A test walks the ordinary pages and asserts none of them carries a `<script>` at all, so "avoided" stays measurable rather than aspirational.

The quick entry is a separate answer to the same problem. `/quick` asks one question per screen, with no navigation and no header, and is installable on a phone's home screen through a web app manifest. It writes through the same `expenses.create` as the long form - same shares, same open bearer, same review queue - because a second way in that books something slightly different is worse than no second way in. Its own script is described above and, like the other one, nothing depends on it.

Money is integer cents, never a float. Expense totals, shares and balances have to reconcile exactly; a float drifts. `money.splitByWeights` uses the largest-remainder method so a split always sums back to the original.

The database runs in a strict SQL mode , applied through the DSN so it reaches every pooled connection. A value a column cannot hold is an error, not a silent truncation. See Data model.

Roles are held per family, not per person. The same person can be the master of their own household and merely a parent in another. Capabilities are explicit sets rather than a rank ladder - see Domain model.

Nothing is created from an identity-provider header. Authelia says who someone is; it never grants them a place in a family's books.

The audit trail is a family's decision, not the installation's. `store/audit.j` records who changed what, and

records nothing at all unless `households.audit_enabled` is set. Two consequences shaped the module: `audit.record` **never throws** - bookkeeping must not fail because the record of it could not be written, so a failure goes to the log and the action stands - and switching the trail off keeps every existing row, because a trail that could be erased by flicking a switch twice is not a trail. Page views are not events; only creations, corrections and deletions are. In `web/app.j` every call site goes through one helper, `noted` , so an entry cannot disagree with the write it describes about who was acting for whom.

Errors are written for the person who will read them , in German, and are shown as they stand. The application has no "an error occurred" page.

Domain model

This is the part to understand before changing anything else.

One rule

The unit of account is the **expense share** : one portion of one payment. Three facts about it are independent:

Fact	Column	Question it answers
who paid	expenses.payer_user_id	who handed over the money
who bears it	expense_shares.bearer_user_id	who carries it in the end
whom it is for	expense_shares.beneficiary_user_id	which child it belongs to

From those, one rule produces every case:

> **A share moves money only when the payer and the bearer are different people, and then the bearer owes the payer.**

Payer	Bearer	Result
child	parent	the parent owes the child
parent	child	the child owes the parent
child	child	nothing owed; still the child's own contribution
parent	parent	nothing owed; the parental contribution the tax office asks about
mother	father	the father owes the mother - the child is untouched

The last row is why the bearer is a *person* rather than a side. Separated parents splitting a child's coat settle with each other exactly as a child settles with a parent, and the child's own position does not move because they bear none of it.

A share may not know its bearer yet

`bearer_user_id` is nullable, and NULL means "a parent, we will work out which". A child recording an expense knows a parent will cover it; which parent is a question for the parents, and making the child answer it recorded a debt against someone who had not agreed to it.

It is a state only a **submitted** expense may be in. Accepting is exactly the moment a parent takes the cost on, so `review` fills the bearer in with whoever approved, and `validateShares` refuses an undecided bearer on anything that skips review. The ledger therefore never sees one: claims are read from accepted expenses only.

Balances

`ledger.balanceOf(personId, counterpartyId, claims, credits)` nets a position.

- `counterpartyId` = `0` nets against **everyone else** - this is a child's wallet.
- A specific counterparty gives the position **between exactly those two** - this is what one parent owes another.

```
netCents    = owedToPerson - owedByPerson - advances + overpaid
walletCents = -netCents
```

`netCents` is the account, positive meaning others owe this person. `walletCents` is the same figure from their side, which is what a child is shown: positive means they are holding money they can still spend.

The wallet

Money moves before there is anything to settle. Parents transfer a lump sum; the child spends it down. Such a payment has no share to attach to, so its unallocated remainder is a `Credit` , and the recipient is holding the payer's money.

Whether an expense draws on the float is decided by who bears it, not by who paid:

```
advance                          +200,00     wallet 200,00
schoolbook 45,00, borne by parent -45,00   wallet 155,00
cinema     15,00, borne by child    0,00   wallet 155,00
```

`ledger.applyCredits(claims, credits)` matches payments to the shares they can discharge. A payment from X to Y can only settle a share that **X bears and Y paid for** - that is the one arrangement where the money already moved the right way. Oldest share, oldest payment first, so the result is reproducible.

This is what makes an expense bought from an advance settled the moment it is accepted, rather than waiting for a reimbursement that already happened.

Dedicated money

A payment may name a **purpose** : one category, on `settlements` (`purpose_category_id`). Its money then clears only shares of expenses in that category, and its remainder waits for one rather than being spent on whatever comes next.

The case it exists for is a rent transfer. 890,00 sent for the rent, treated as an undifferentiated float, is eaten piecemeal by whatever else is open - and the books then say that transfer bought schoolbooks, which is wrong on its face and worse in front of a tax office.

Two rules follow, both in `ledger.applyCredits` :

- a credit with a purpose is offered only claims whose `categoryId` matches (`ledger.creditCovers`);
- **dedicated credits are offered before general ones.** If the general float paid a rent share, the rent money would be left with nothing it is allowed to buy and would never be spent at all.

The match is on the expense's category **as it stands now** , which is the same category the reports read. Re-categorising an expense therefore moves it into or out of reach of dedicated money, and the callers release and re-apply so the two cannot drift apart.

Approval

A child records what they paid and asks to be reimbursed; an adult accepts or rejects it. Only `accepted` expenses reach a balance - `submitted` is still a request, `rejected` never counts.

An expense that **asks nobody else to pay** is accepted on the spot, whoever recorded it. A child noting what they bought from their own money is telling the family what they spent, not asking for anything, so it does not sit in the parents' queue. See `expenses.asksAnyoneElseToPay` .

Locking

Once money has moved **for an expense** , it is frozen (`expenses.isLocked`). Otherwise a child could be reimbursed and then edit the entry to say the parents were the payer, leaving a payment on record that matches nothing.

- The **financial** fields - amount, split, payer, date - cannot change.
- The **bookkeeping** fields - category, tax relevance, description, note - can, through `expenses.reclassify` . That is the tax adviser's working surface, and it must not require unwinding a payment that has already happened.
- A child may edit their own entry only while it is still `submitted` .

An allocation out of a float is not money moved for that expense. The transfer happened before the entry existed and was not about it; the application decided the two belonged together. So such an allocation carries `settlement_items.auto_cents` , `isLocked` is `settled` > `auto` , and the entry stays fully correctable: `settlements.releaseAutomatic` gives the money back, the correction is written, and `applyAvailableCredit` settles whatever it now comes to. Every write path that touches an expense does this - correcting, re-categorising, rejecting, deleting.

Getting this wrong the other way is what the wallet model originally did: an expense bought from an advance

locked on acceptance, so a mistyped amount could only be fixed by reversing the whole advance.

Roles and capabilities

Permissions are **not a ladder** . An adviser needs the Finanzamt reports, which a parent does not get, while being barred from settling, which a parent does. No ordering expresses that, so each capability names the roles that hold it.

	child	parent	master	advisor	admin
record own	yes	yes	yes	-	yes
record for children	-	yes	yes	yes	yes
edit entries	-	yes	yes	yes	yes
delete entries	-	yes	yes	-	yes
view all children	-	yes	yes	yes	yes
accept / reject	-	yes	yes	-	yes
settle payments	-	yes	yes	-	yes
Finanzamt reports	-	-	yes	yes	yes
manage members	-	-	yes	-	yes
administer installation	-	-	-	-	yes

`roles.sortOrder` exists only to order a member listing; **no permission is derived from it** . A test pins down that capabilities are not a ladder, so a future refactor back to ranks fails loudly.

Income, and the two questions it answers

An income row carries two flags, because the FLAG asks two different things about a child's money and the answers differ:

	taxable	own_income
what it feeds	Zuverdienstgrenze, § 5 Abs. 1	überwiegende Kostentragung, § 2 Abs. 2
wages	yes	yes
Lehrlingsentschädigung, Waisenpension	no (§ 5 excludes them)	yes
Wohnbeihilfe, Studienbeihilfe the child applied for	no (§ 3 EStG: tax-free)	yes
Familienbeihilfe	no	no

`taxable` is asked on the form, defaulted from the kind, because the same word covers both answers: a Studienbeihilfe under the StudFG is tax-free, a private scholarship of the same name may not be.

`own_income` is *derived* from the kind and stored, never asked. It is false for exactly one kind, `family_allowance` , and that is the point of the kind existing: § 2 Abs. 6 reduces the cost of maintenance by what **the child** receives, and the Familienbeihilfe is granted to the entitled parent (§ 2 FLAG). § 14 direct payment moves the money, not the entitlement. Counting it as the child's own would shrink the very need the parents are trying to show they carried - the family would be arguing against itself out of its own books.

The same fact has a second consequence, in the other direction. A child who pays a bill out of that Beihilfe is spending the parents' money, so `ChildReport.allowanceBorneCents` moves that much from the child's side to the parents' before `coverage.analyze` sees it - capped at what arrived in the month, computed per month, and only where the family set the child's Eigenbeitrag to `family_allowance` , which is the statement that the money reaches her at all. The household view keeps the raw split: she did pay those bills.

`incomes.totalFor` is what arrived, whosever it is, and feeds the income page. `incomes.ownTotalFor` is the § 2 Abs. 6 figure and feeds `ChildReport` . `incomes.taxableTotal` is the § 5 Abs. 1 figure and feeds

`limitCheck` .

Messages, and mail as a copy

`inbox_messages` is one row per person per event. Every notice writes one; mail is sent afterwards and only where `config.mailConfigured` and an address exist. The order is the design: an installation without SMTP could previously say nothing at all, and mail is not a record - delivered or not, filtered or not, read or not, and never knowable. `read_at` holds the **first** reading and never moves; the question it answers is "did this reach them", which has one answer per message.

`notices.deliver` is the single place both channels are written, so an event cannot reach one and miss the other. The per-family switches (`notifications`) gate both: they were always "what do you want to hear about", and that is now one question rather than two.

Payments, and who says they arrived

`settlements.receipt_state` is `pending` , `confirmed` or `missing` , answered only by `to_user_id` - a payer confirming their own transfer is the same signature twice. Three states and not a flag, because "not asked yet" and "the recipient says it never came" are different facts.

The ledger does not wait for it. A payment settles what it settles when it is booked; a dispute is a red row on the parents' overview and `reverse` is still the deliberate second act that moves money back. Making the books depend on a child's tap would leave a family whose child is on a school trip unable to see where it stands.

Coverage

The Austrian family-allowance rule, in `coverage.j` :

```
netNeed  = max(0, expenses - income)
required = min(netNeed, netNeed * threshold / 100 + 1)   # integer division
met      = netNeed > 0 and parentBorne >= required
```

The denominator is the remaining need , not the gross spend: the child's own income is set against the cost of maintenance before anybody's share is measured. § 2 Abs. 6 FLAG says so outright for tax-free receipts - the cost of maintenance is to be taken *reduced by* them - and the case law applies the same comparison generally, putting the child's living costs on one side and their own income (taxable and tax-free) together with the claimant's payments on the other. Measuring against the gross spend would count the child's own money as a cost somebody else still has to carry.

The comparison is **exact integer arithmetic** , never a rounded percentage: a case sitting one cent below half must not pass because it displays as 50,0 %. And *on* the threshold is a miss - **überwiegend** is strictly more than half, so `required` is the first cent past the share and carrying exactly half fails by one cent. The `min` is there because nothing can exceed the whole: a threshold of 100 % is read as "all of it" rather than as a test no family could pass.

`netNeed` == `0` is a state of its own, and the interesting one. The child's income reached everything recorded, so **nobody can carry the greater part of what is left** , however much the parents actually paid: 554,60 of receipts against 3.631,60 of income is a gift and not maintenance. `meetsThreshold` is false, `selfSupporting` is true, `requiredCents` and `shortfallCents` are 0 - there is no amount that would have fixed it - and the interfaces show a neutral badge with the reason rather than a red one, because the family did nothing wrong.

`coveragePermille` is the share of the *remaining need* and therefore routinely exceeds 1000: parents who paid the bills while the child's income stayed in the child's pocket are over 100 %. `cappedPermille` exists for the bar.

`expenses` == `0` is the case with no answer at all: nothing recorded proves nothing in either direction, so `needApplicable` is false and the report says *Nichts erfasst* rather than drawing a bar.

Three outcomes, and a report has to tell them apart:

	needApplicable	selfSupporting	meetsThreshold

ordinary period	true	false	the comparison above
income covers every expense	true	true	false
nothing recorded at all	false	false	false

Neither of the last two is a finding against the family - one is an empty period, the other a child who can pay their own way - so both get a neutral badge and a sentence, never the red one.

Data model

MariaDB, InnoDB, `utf8mb4` . Migrations live in `src/migrations.j` and are applied by `sqlmigrate` , which records them in `schema_migrations` .

Tables

Table	Holds
households	a family
users	a person; username is the identity provider's Remote-User
household_members	who is in which family, and as what
categories	a family's expense categories, with a tax-relevance default
expenses	one payment
expense_shares	one portion of a payment: beneficiary, bearer, amount
settlements	one bank transfer
settlement_items	which share a transfer paid, and by how much
incomes	a child's own income, per month
receipts	attachments, with extracted text
bank_imports	a record of each CSV import
notification_settings	who wants to be written to about what, per family
mail_queue	messages waiting to be sent, and what became of them
audit_log	who changed what, for the families that switch it on

The joins that carry the model

users has no household. A person is global and unique by `username` ; `household_members` carries the per-family role. That is what lets a child belong to two families and a person hold different roles in each.

expense_shares names two people. `beneficiary_user_id` (nullable) is the child the portion is for; `bearer_user_id` is who carries it. Combined with `expenses.payer_user_id` , those three columns produce every settlement case - see Domain model.

notification_settings hangs off the pair, the address off the person. `users.email` is the mailbox - one person has one, whichever families they are in - while what they want to hear about is a row per (`household_id` , `user_id` , `kind`). Somebody may want everything about the family they run and nothing about the one they are merely a parent in. A **missing row means the class default** , not "off", so a class added to `core/notify.j` later starts at its default for everyone rather than arriving switched off for exactly the people who once saved their settings.

settlement_items is what a transfer paid. A transfer that covers five expenses has five items. The part of a transfer with no items is an advance, and that subtraction is the whole of the wallet.

auto_cents is the part of an item nobody booked. A float settles a new expense the moment it is accepted; `settlements.applyAvailableCredit` records what it allocated in `auto_cents` as well as in `amount_cents` . Everything that asks "has money moved for this expense" compares the two: `expenses.isLocked` is `settled` > `auto` , and `hasSettlement` looks for `amount_cents` > `auto_cents` . So an entry paid out of an advance stays correctable - the allocation is given back by `settlements.releaseAutomatic` and re-applied afterwards - while an entry somebody actually paid for is as frozen as it ever was. Without the split, fixing a typo in a 12,00 expense meant reversing the 890,00 transfer the float came from.

settlements.purpose_category_id dedicates money to one kind of cost. A payment with a purpose clears only open shares of expenses in that category, and its remainder waits as a credit only that category may draw

on (`ledger.creditCovers`). Dedicated credits are also offered to the allocator *before* general ones: if the general float paid a rent share, the rent money would be left with nothing it is allowed to buy. The match is on the expense's current category, so re-categorising an expense moves it into or out of reach and the callers re-run the allocation.

audit_log is per family and off by default. `households.audit_enabled` decides whether `audit.record` writes anything at all, so a family that has not asked for a trail carries no rows. Both user columns are nullable and their foreign keys are `ON DELETE SET NULL` : a deleted account must not take the record of what it did with it. A NULL `actor_user_id` therefore means *no account stands behind this change* - one since removed, or something done with `familienkonto` at the shell, where the summary ends in `(Kommandozeile)` . `acting_for_user_id` is set when somebody was acting as another person, which is the one case where "who did this" has two answers. `summary` is German prose written **at the time of the change** rather than rebuilt from ids on display - rebuilding it later would describe the books as they are now, which is precisely the question the log exists to answer.

Columns worth knowing about

Column	Why it exists
expenses.amount_cents	the base-currency amount; the only one any balance or report reads
expenses.original_amount_cents / original_currency / exchange_rate	a foreign-currency receipt kept auditable; never used in arithmetic
expenses.status	submitted / accepted / rejected; only accepted counts
expenses.tax_relevant	independent of approval: repayable is not the same as tax-relevant
expenses.evidence + linked_expense_id	a cost that is proven by another entry's receipt rather than its own
expenses.external_ref	the bank's transaction reference; UNIQUE(household_id, external_ref) deduplicates imports, and repeated NULLs are permitted so manual entries are unconstrained
users.email	optional; empty means the person is written to nowhere
receipts.checksum	SHA-256, re-verified on read so an altered file is refused
receipts.sha512	the content hash; two identical uploads share one stored file
household_members.self_contribution_kind	the kind of the child's own income that goes towards their own costs; the amount is whatever they recorded for that month, never a second copy
households.audit_enabled	whether this family keeps an audit trail; off unless asked for
settlements.purpose_category_id	the category this money is dedicated to; NULL is the ordinary undedicated payment
settlement_items.auto_cents	the part of the allocation the application made out of a float, which can be given back
audit_log.entity + entity_id	what was touched; entity_id is NULL once the thing has no id any more
receipts.extracted_text + text_source	the document's text and how it was obtained (plain / pdftotext / ocr)

receipts.kind	receipt / invoice / bank_statement / payment_slip / other; a payment slip proves one transfer, a statement covers a month of them

Exchange rates are integers in **millionths** (`1.085432` -> `1085432`), so a conversion is exact and reproducible.

Strict SQL mode

Every connection runs under:

```
STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_ENGINE_SUBSTITUTION
```

This is applied **through the DSN** , not with a `SET SESSION` after connecting. Go's `database/sql` keeps a connection pool and hands out whichever connection is free, so a statement run on one says nothing about the others. A DSN parameter is applied by the driver to every connection it opens.

`db.withStrictMode` appends it to whatever DSN you configure, leaving an explicit `sql_mode=` alone. `db.isStrict` lets a caller verify rather than assume; `familienkonto doctor` reports it.

Without it, an over-long merchant name would be silently truncated and a malformed amount would become zero. For accounting data that is not a preference.

> **Do not put parseTime=true in the DSN.** It makes the driver return DATE columns as RFC 3339 timestamps. Dates are handled as plain strings throughout; `db.dateOf` normalises defensively.

Transactions

`db.begin` / `commit` / `rollback` / `executeIn` / `insertIn` . Used where a write is only meaningful as a unit - a settlement and the items recording what it paid must both exist or neither, or a balance is quietly wrong with nothing to detect it. `db.rollback` is safe as an `errdefer` .

Migrations

```
familienkonto migrate status
familienkonto migrate up
familienkonto migrate down --steps 1
```

Versions are zero-padded and applied in lexical order, each in its own transaction. `migrate up` is idempotent.

A new household is given the standard categories by `users.createHousehold` itself - a family without categories cannot record a single expense, so it is not left for someone to notice later.

That was not always true, and migration `003` is the consequence. `002` seeded a household called *Familie* purely so the default categories had an owner. Once `createHousehold` began seeding its own, the seeded household became a family that belonged to nobody and held nothing - visible as an empty row in every list of families, including the administration page. `003` removes it.

Every statement in `003` is guarded on the household still having the name `002` gave it and having nothing at all hanging off it: no members, expenses, incomes, settlements or imports. An installation where somebody has adopted id 1 as a real family keeps it, and the migration still counts as applied - it is a tidy-up, not a demand. `src/migrations_test.j` covers both outcomes.

Authentication

Three modes, selected by `APP_AUTH_MODE` :

Mode	Who authenticates	Needs a proxy
dev	nobody: pick a name from a list	no
authelia	a forward-auth proxy, believed through headers	yes
oidc	the application itself, against an OpenID Connect provider	no

`authelia` and `oidc` are both production modes and both work with Authelia - they differ in *who does the talking* . Pick `authelia` when a proxy already authenticates everything on that host, and `oidc` when you would rather the application signed people in itself.

Production: Authelia forward-auth

A reverse proxy authenticates the request against Authelia and passes the result on as headers:

```
Remote-User    the username
Remote-Email   the mail address
Remote-Name    the display name
Remote-Groups  comma-separated groups
```

Those headers are believed only when the request came from a configured proxy. This is the whole of the security of the arrangement. A header is just text: a request arriving from anywhere else claiming `Remote-User: admin` is anonymous, and there is a test that fires exactly that attack.

`APP_TRUSTED_PROXIES` takes addresses and CIDRs, IPv4 and IPv6, with the port stripped. **A malformed entry is ignored rather than matched** , so a typo narrows access instead of opening it. An installation in `authelia` mode that names no trusted proxy is refused at startup by `config.fromMap` .

What Authelia decides

Two things, and no more:

Setting	Effect
APP_GROUP_ACCESS	must be held to use the application at all; empty admits everyone Authelia lets through
APP_GROUP_ADMIN	grants the installation-administrator role

The role someone holds **in a family** comes from `household_members` . That is what lets one person be the master of their own family and a parent in another, which no group mapping could express.

No account is created from a header

Authelia may know someone this application does not. They are told so clearly; they do not get an account, and they certainly do not get a place in a family's books. Accounts are created by the family's master or by an administrator.

Production: OpenID Connect

`APP_AUTH_MODE=oidc` makes the application an OIDC **client** : no proxy, no headers, no `APP_TRUSTED_PROXIES` . It sends the reader to the provider, the provider sends them back with a code, and the application exchanges that code for the claims on a back channel it opens itself.

The authorization-code flow with a confidential client, PKCE (S256) on every attempt, `state` and `nonce` on every attempt:

```
/login          the button
/oauth2/login   mints state, nonce and a PKCE verifier, redirects to the provider
/oauth2/callback   checks state, exchanges the code, reads the claims, signs in
```

The callback URL is APP_BASE_URL + /oauth2/callback - that is the value a provider wants under `redirect_uris` .

The two have to match **character for character** , and a provider that is unhappy says so in a way that names the parameter but not what it expected:

> `invalid_request` - The 'redirect_uri' parameter does not match any of the OAuth 2.0 Client's pre-registered 'redirect_uris'.

The usual causes, in order of how often they are the answer:

- **a different path** - `/oauth2/oidc/callback` (Authelia's own documentation example) against this application's `/oauth2/callback` ;
- **http against https** , or a host that differs by a `www.` ;
- **a trailing slash** on one side;
- **APP_BASE_URL pointing at the container** (`http://app:8080`) rather than at the address people type.

`fk doctor` prints the exact value to register, and so does the log line the application writes at startup in `oidc` mode:

```
OIDC: issuer https://sso.example.com, client familienkonto,
      redirect_uri https://familienkonto.example.com/oauth2/callback
```

If the provider's registration cannot be changed, set **APP_OIDC_REDIRECT_URL** to the URI it already has. The application then tells the provider that URI *and serves the callback at its path* - so `/oauth2/oidc/callback` works as well as the default, and the setting is not a way to configure a 404.

Setting	
APP_OIDC_ISSUER	the issuer URL; everything else comes from its /.well-known/openid-configuration
APP_OIDC_CLIENT_ID / APP_OIDC_CLIENT_SECRET	the registered client; the secret is sent with HTTP Basic
APP_OIDC_REDIRECT_URL	usually empty - derived from APP_BASE_URL
APP_OIDC_SCOPES	default openid profile email groups
APP_OIDC_USERNAME_CLAIM	default preferred_username
APP_OIDC_GROUPS_CLAIM	default groups
APP_OIDC_CREATE_USERS	create the account on a first sign-in (default true)

APP_SECRET is required as well: it signs the session and the short-lived cookie that carries `state` , `nonce` and the PKCE verifier while the reader is away at the provider. That cookie is why a restart, or a second replica answering the callback, does not lose somebody mid-login.

The Authelia side

This is the client entry Authelia wants - and unlike forward-auth, here it really is needed:

```
identity_providers:
  oidc:
    clients:
      - client_id: familienkonto
        client_name: Familienkonto
        client_secret: '$pbkdf2-sha512$310000$...'   # the hash; the app gets the
plain secret
        public: false
        authorization_policy: one_factor
        require_pkce: true
        pkce_challenge_method: S256
        redirect_uris:
          - 'https://familienkonto.example.com/oauth2/callback'
        scopes: [openid, profile, email, groups]
        grant_types: [authorization_code]
        response_types: [code]
        token_endpoint_auth_method: client_secret_basic
```

`require_pkce: true` is safe: the application always sends an S256 challenge.

What is checked, and what is not

The ID token's signature is **not** re-verified, and that is deliberate rather than missing. The token is not read out of the browser: it arrives on a TLS connection this process opened to the issuer's own token endpoint, authenticated with the client secret, in answer to a code minted seconds earlier - the case OpenID Connect Core §3.1.3.7 exempts. What *is* checked is everything an attacker could otherwise influence:

- the **state** must equal the one in our signed cookie, or the callback is not the answer to a request this application made;
- the **iss** must be the issuer from discovery;
- the **aud** must contain our client id;
- the **exp** must be in the future;
- the **nonce** must equal the one generated for this attempt;
- the sign-in must have started less than ten minutes ago.

If you would rather have JWKS signature verification as well, that is a change to `src/app/oidc.j` and nothing else.

The first sign-in creates the account

Unlike forward-auth mode, `oidc` **provisions** : somebody the provider authenticates, who holds `APP_GROUP_ACCESS` , and who has no account here yet gets one on their first sign-in, with the **display name** and the **mail address** from the claims. They belong to **no family** - which family is a decision about a family's books, not about an identity - so they land on the page that says so, and a master or an administrator adds them.

```
oidc: mplx, name "Martin Maier", email "martin@example.at", groups: app_
familienkonto
auth: created account mplx (Martin Maier) from the identity provider
```

Set `APP_OIDC_CREATE_USERS=false` for an installation that would rather make every account by hand with `fk user-add` ; the refusal is then *"Für … ist in dieser Anwendung noch kein Zugang eingerichtet"* .

While provisioning is on, **the administration stops offering to create accounts** : *Administration › Benutzer* says where accounts come from instead of showing the form, and a hand-made POST is refused too. A username typed there is a guess at what the provider will send, and a wrong guess makes a second account beside the one the first sign-in creates.

What the administration is still for: giving somebody a family, correcting the **display name** and the address,

locking an account, and granting or withdrawing administration. The name matters more than it looks - a provider that sends no `name` claim leaves the account called after its username, and nothing overwrites it afterwards, so it is corrected here or on the person's own *Einstellungen* page.

APP_GROUP_ACCESS is what decides who gets one , and it is checked before anything is written. With provisioning on and no access group configured, every account the provider authenticates would get one - the application says so at startup rather than letting that be discovered later:

```
AUTH: an account is created on first sign-in and APP_GROUP_ACCESS is empty -
everybody your identity provider authenticates gets one.
```

On **later** sign-ins the provider fills in what is missing and changes nothing else: an account without an address adopts the one from the claims, an address somebody corrected in the application stays corrected, and the display name a family typed is never overwritten. The same rule the family page follows when it adds a person who already has an account.

`APP_GROUP_ADMIN` **grants** the administrator flag and never withdraws it, which is the rule forward-auth mode follows too. Withdrawing is done under *Administration › Benutzer* . Anything else would demote an `--admin` account on its very first sign-in, before any group existed at the provider.

Reading the log

Every sign-in writes one line with everything the next decision is made from, which is the fastest way to see what a provider is actually sending:

```
oidc: mplx, name "Martin Maier", email "martin@example.at", groups: app_
familienkonto, app_familienadmin
```

`groups: (none)` means the claim did not arrive: check that `groups` is in the client's `scopes` at the provider *and* in `APP_OIDC_SCOPES` , and that the provider's userinfo is not signed (Authelia: `userinfo_signed_response_alg: 'none'`) - a signed userinfo document reads as empty here, and the groups would then have to come from the ID token alone.

Forward-auth has no callback URL

In authelia mode there is no `client_id` , no client secret, no token exchange and no redirect URI - the application never talks to Authelia at all. It reads four headers from a proxy it trusts, and that is the entire protocol. (In `oidc` mode there *is* a callback, and it is the section above.)

So an entry like this under `identity_providers.oidc.clients` configures nothing and can be deleted:

```
# NOT needed - the application is not an OIDC client
- client_id: familienkonto
  client_secret: '$pbkdf2-sha512$310000$...'
  redirect_uris:
    - 'https://familienkonto.example.com/oauth2/oidc/callback'
```

There is no route at that path, and a request to it gets the 404 page. What Authelia needs instead is an `access_control` rule for the domain (below); the proxy does the rest.

The only Authelia URL the application knows is `APP_AUTHELIA_URL` , and it is not a callback either: it is where the *Anmeldeportal* button points after signing out, so somebody can end the Authelia session as well as this one.

Forward-auth is the deliberate choice. It keeps every credential, every second factor and every session out of this application - there is no token to leak here, and no login form to attack - and it is what lets the same deployment sit behind Authelia, Authentik or a plain `auth_request` without knowing which.

Configuring Authelia and the proxy

Three pieces have to agree: Authelia protects the application, the proxy asks Authelia and forwards the answer, and this application trusts that proxy and nobody else.

1. Authelia

Nothing special is needed beyond an access rule and, if you want to use the groups, groups on the users. A minimal `configuration.yml` :

```
access_control:
  default_policy: deny
  rules:
    - domain: familienkonto.example.com
      policy: two_factor          # one_factor is enough for a family
      subject:
        - "group:expenses"        # everybody who may use it at all

session:
  cookies:
    - domain: example.com         # the parent domain of both hosts
      authelia_url: https://sso.example.com
```

The **session cookie domain has to cover both hosts** - the portal and the application - or the browser will not send the session to Authelia when the proxy asks.

Groups are ordinary Authelia groups, in `users_database.yml` or your LDAP:

```
users:
  martin:
    displayname: "Martin Maier"
    email: martin@example.com
    groups:
      - expenses            # APP_GROUP_ACCESS
      - expenses-admins     # APP_GROUP_ADMIN
```

Family roles are **not** groups. `expenses-admins` administers the installation; who is a parent or a child in which family lives in `household_members` , because one person is regularly a master in their own family and a parent in another.

2. The proxy

The proxy calls Authelia for every request and copies four headers from the answer onto the upstream request. **It must also delete any Remote-* header the client sent itself** - otherwise anybody can claim to be anybody, and the trusted-proxy check cannot help, because the request really did arrive from the proxy.

Caddy:

```
familienkonto.example.com {
    forward_auth sso.example.com {
        uri /api/authz/forward-auth
        copy_headers Remote-User Remote-Email Remote-Name Remote-Groups
    }
    reverse_proxy app:8080 {
        header_up -Remote-User
        header_up -Remote-Email
        header_up -Remote-Name
        header_up -Remote-Groups
    }
}
```

(`forward_auth` sets the four headers *after* those deletions, which is the order that makes this safe.)

nginx:

```
server {
    server_name familienkonto.example.com;

    # Whatever the client sent is not evidence.
    proxy_set_header Remote-User "";
    proxy_set_header Remote-Email "";
    proxy_set_header Remote-Name "";
    proxy_set_header Remote-Groups "";

    location /authelia {
        internal;
        proxy_pass http://authelia:9091/api/authz/auth-request;
        proxy_set_header X-Original-URL $scheme://$http_host$request_uri;
        proxy_set_header Content-Length "";
        proxy_pass_request_body off;
    }

    location / {
        auth_request /authelia;
        auth_request_set $user  $upstream_http_remote_user;
        auth_request_set $email $upstream_http_remote_email;
        auth_request_set $name  $upstream_http_remote_name;
        auth_request_set $groups $upstream_http_remote_groups;
        proxy_set_header Remote-User $user;
        proxy_set_header Remote-Email $email;
        proxy_set_header Remote-Name $name;
        proxy_set_header Remote-Groups $groups;

        error_page 401 =302 https://sso.example.com/?rd=$scheme://$http_
host$request_uri;
        proxy_pass http://app:8080;
    }
}
```

Traefik (labels on the application container, with an Authelia middleware):

```
labels:
  - "traefik.http.middlewares.authelia.forwardauth.address=http://authelia:9091/
api/authz/forward-auth"
  - "traefik.http.middlewares.authelia.forwardauth.authResponseHeaders=Remote-
User,Remote-Email,Remote-Name,Remote-Groups"
  - "traefik.http.routers.familienkonto.middlewares=authelia@docker"
```

3. This application

```
APP_AUTH_MODE=authelia
APP_TRUSTED_PROXIES=10.0.0.0/8,172.16.0.0/12   # where the proxy talks from
APP_AUTHELIA_URL=https://sso.example.com/      # only for the "sign in" link
APP_GROUP_ACCESS=expenses                      # empty admits everyone Authelia
lets through
APP_GROUP_ADMIN=expenses-admins
APP_BASE_URL=https://familienkonto.example.com
```

`APP_TRUSTED_PROXIES` is the address the **proxy** connects from, as this application sees it - in Docker that is the container network, not the address of your router. `fk doctor` prints what the process is configured with; if

every request comes out anonymous with *"nicht von einem vertrauenswürdigen Proxy"* , that list is what is wrong.

4. Then create the accounts

Authelia knowing somebody does not give them an account here. The first one is made on the command line, the rest in the interface:

```
bin/fk user-add martin --name "Martin Maier" --admin
```

The username must be **exactly** what Authelia sends as `Remote-User` .

Development: passwordless login

`APP_AUTH_MODE=dev` offers a list of users to pick from. The choice travels in a cookie signed with `APP_SECRET` (HMAC-SHA256, compared constant-time), so it cannot be edited into somebody else - a test forges exactly that.

Guards:

- `auth.devUsers` and `auth.devLogin` **throw** in `authelia` mode, so the passwordless path cannot be reached on a production installation.
- Forward-auth headers are ignored entirely in `dev` mode.
- `auth.startupWarning` returns a blunt warning, printed by the server at startup and by `familienkonto doctor` .

Arriving without a session

Any page asked for without a session answers **303 to /login** , whatever the mode. There is no "you are not signed in" page in between: it said the same thing as the login page it linked to, and under OIDC that made three screens - not signed in, then sign in here, then the provider - before anybody had typed anything.

The login page carries the *reason* when there is one worth carrying: a locked account, an unknown username, a request that did not come through the proxy. Simply having no session is not one - everybody starts there - so a first visit is the sign-in button and nothing else.

Session lifetime

authelia mode	Authelia's session decides. This application keeps no session of its own: it reads the forward-auth headers afresh on every request, so when Authelia stops vouching for somebody, the next click is anonymous.
dev mode	The signed cookie is good for 24 hours (auth.SESSION_SECONDS).
Impersonation	One hour (auth.ACT_AS_SECONDS), whichever mode.

The lifetime is **inside the signature** , not only in the cookie: `mintSession` stamps the moment it was made and `readSession` refuses a token older than the limit. A cookie `maxAge` is a request to the browser and nothing more - a token copied out of one would otherwise have been valid for ever, which is exactly the case a timeout is for. A stamp from the future is treated as expired rather than trusted: the only ways to get one are a clock that jumped or a token somebody has played with.

There is no idle timeout and no server-side session store: the token is self-contained, so signing out is a matter of clearing the cookie, and there is nothing to garbage-collect.

Identity

`auth.identify(conn, cfg, request)` returns an `Identity` . The `Request` struct carries only plain strings, so every rule is testable without starting a server.

- `auth.hasFamily` - authenticated **and** placed in a family with a role. Someone authenticated who belongs to no family is a real state, and the interface says so rather than showing an empty ledger.

- `auth.withHousehold` - switch families. A household the person does not belong to **leaves them where they were** rather than turning them out of their own: a tampered id in a request should be ignored, not become a way to lock someone out.
- `auth.mayViewChild` - a child sees their own data; everyone else in the family sees every child.

In the web layer

Every handler works the same way: establish the identity, refuse what the role does not carry, then render. Refusals are never left to the interface omitting a link - the navigation hides what a role cannot use, *and* the handler refuses it.

CSRF

`web.csrfToken` mints a token **and sets a cookie** , so calling it twice in one request invalidates whatever the first call produced.

This application does not use it for a signed-in reader. Per-request tokens have a worse problem than per-form ones: the cookie is replaced on *every* page load, so a page left open in another tab carries a token that no longer matches. Pressing "Abmelden" there answered "Die Sitzung ist abgelaufen" while the session was alive, and left the reader with no way out.

`csrfToken` in `web/app.j` therefore derives the token from the account - an HMAC over the user id - so it is the same on every page for as long as the sign-in lasts, unguessable without `APP_SECRET` , and different per person. The id in the payload is not the secret part; the signature is. It is deliberately not the session cookie, which would put a `HttpOnly` value into the HTML.

Anonymous requests keep the per-request token: the login page has nothing to bind to, and one fixed token for every visitor could be used to forge the sign-in POST.

The request body can be read only once, and `web.csrfCheck` reads it looking for the field. A handler that also needs the other fields therefore reads the form itself and passes it to `formCsrfOk` . Multipart uploads carry their own check, because `web.csrfCheck` only reads the form field for an urlencoded body.

That "once per request" reaches further than the handler's own forms: the page chrome carries a form too. The sign-out button in the top bar is a POST, so `views.page` takes the request's token as a parameter and `show` passes it through - a chrome that minted its own would overwrite the cookie *after* the body had been rendered, and break every form on the page at once.

`tests/login_test.j` holds that invariant: it drives a real server and asserts that a page carries exactly one distinct token and that it matches the cookie the response set.

`POST /household` (switching family) and `POST /import/preview` (the CSV preview) were missing the check and now carry it - the preview because a multipart body carries its token as a part, so `web.csrfCheck` cannot find it and the handler has to compare it itself.

Acting as another user

An administrator can work through the interface as somebody else - the way to reproduce "my balance is wrong" without asking for a password. A managing parent can do the same for **the children of their own family** , from `/family` , because most installations have one administrator and every family has a managing parent. It is a second, separate cookie (`expenses_actas`), not a second session:

- the token carries **both ids** , the taker's and the target's, so a cookie lifted from one account is not usable by another;
- the token also carries **the family the takeover is confined to** , and for a managing parent that confinement is the whole of the safety: `withHousehold` refuses to move a pinned identity, the family switcher is not rendered, and `?household=` does nothing. Without it, taking over a child of separated parents would have handed the taker the *other* family's books - the child sees them, so the takeover would too;
- who may take whom over is decided on **every request** , not once when the cookie was handed out: an administrator may take over anybody, anywhere; a managing parent may take over an account that is a

child in the pinned family , of which they are the master. An account that administers the installation is refused outright, because the way back up must never run through somebody else's login;
- an earlier rule demanded that the taker manage *every* family the target belongs to. It was safe, and it also meant that in a patchwork family - which this application models deliberately - neither parent could ever use the feature at all. Pinning replaced it;
- a takeover cannot be started while one is already running: a second cookie would overwrite the first and leave the administrator who started it with no way back;
- the resulting identity is **entirely the target's** - their family, their role, their permissions. Administration is not carried along, so while acting as a child the administration page is as closed as it is for that child;
- the identity keeps `realUserId` and `realDisplayName` , which is what lets every page say who is really there and offer the way back;
- it expires after an hour, and `POST /admin/release` ends it sooner. That route deliberately needs no rights of its own: the identity holding the cookie is the impersonated one, and whoever holds it must always be able to put it down. It sends the reader back where the takeover started - the administration for an administrator, `/family` for a managing parent, who would only be refused by the other one.

`src/app/auth.j` decides all of this in `actingAs` and `mayActAsMember` ; `src/app/auth_test.j` , `tests/administration_test.j` and `tests/family_test.j` cover the refusals, which are the part worth testing.

Signing out

`POST /logout` only, and it checks the token. There is deliberately no `GET` route: a URL that ends a session can be fired by anything that makes a browser fetch it, and being signed out by someone else's image tag is a nuisance no reader can explain. The button is rendered on every page a signed-in reader can reach, including the 403 and 404 pages, so a wrong-account sign-in is always one click from being undone.

Deployment

Quick start

```
cp .env.example .env      # change the passwords first
docker compose up -d      # database, migrations, application
```

The application is then on http://localhost:8080 . `docker compose up` runs `migrate` to completion before starting `app` .

The image

Built on the official Jennifer interpreter image:

```
ARG JENNIFER_IMAGE=ghcr.io/jennifer-language/jennifer:main
```

The sources declare `# pragma-jennifer-version: >=0.25.0` . No released interpreter satisfies that yet - 0.24.0 is the newest tag - and the pragma is a hard abort, so a released image would refuse to run the application at all. The `main` build reports itself as a development version, and a development build bypasses the floor. **Pin this to :0.25.0 as soon as that release exists.**

The Debian-based variant is used rather than the distroless one because the Finanzamt export shells out to PDF tools, which need a shell.

Building it

Use `scripts/build-image.sh` , which stamps the image with what it was built from - the image carries no `.git` , so the version has to be read at build time and passed in:

```
scripts/books.sh                # the handbooks first; the COPY needs them
scripts/build-image.sh          # -> app-expenses:latest
scripts/build-image.sh registry.example.com/familienkonto:2026-09-02
```

A tag **on this commit** becomes the version; anything else leaves the version empty and sets only the commit, and a dirty tree appends `+dirty` - a working copy is not the commit it claims to be. The settings page then reads

```
Familienkonto 1.2.0                     (built from a tag)
Familienkonto (Entwicklung, 0e173c1)    (built from a commit)
Familienkonto (Entwicklung)             (built by hand, or run from a checkout)
```

and the same two values land in `org.opencontainers.image.version` and `.revision` , so `docker inspect` answers the question without starting the container. It is the one line that makes a bug report answerable.

`docker build` by hand still works and simply says *Entwicklung* .

The handbook has to be rendered first: the image copies `book-output/` in, and the `COPY` fails without it - deliberately, so a missing manual is a build error rather than a dead *Handbuch* link in a running container.

```
scripts/books.sh                                # writes book-output/
docker build -f docker/Dockerfile -t app-expenses:latest .
```

With OCR, which is the only optional tool:

```
docker build --build-arg WITH_OCRMYPDF=true \
  -f docker/Dockerfile -t app-expenses:ocr .
```

That is **749 MB** against 436 MB for the default build; tesseract and ghostscript are the difference. Through compose the switch lives in `.env` (`WITH_OCRMYPDF`) and is read by `docker compose build` .

Installing `ocrmypdf` does not switch OCR on: `APP_OCRMYPDF_BIN` is empty by default, so set it to `ocrmypdf` at runtime as well. The binary being present and the feature being armed are two decisions, and the second one

belongs to the installation rather than to the image.

For a swarm or any multi-node setup, build once and push - the nodes cannot build, and `latest` on three of them is three versions:

```
docker buildx build --platform linux/amd64,linux/arm64 \
  --build-arg WITH_OCRMYPDF=true \
  -f docker/Dockerfile -t registry.example.com/familienkonto:2026-09-01 --push .
```

`pdfcpu` is fetched from its release tarball and the Dockerfile maps `TARGETARCH` onto the right one, so amd64, arm64, armv7 and i386 all build.

PDF tooling

Tool	Role	Installed
pdfcpu	primary merge engine	always, from its release tarball
qpdf	merge fallback, repair, linearise	always
pdftotext	reads text out of PDFs	always (poppler-utils)
ocrmypdf	OCR for photographed receipts	only with --build-arg WITH_OCRMYPDF=true
magick	the smaller copy of a photograph a browser is sent	always (imagemagick)

OCR is opt-in because it pulls in tesseract and ghostscript and roughly triples the image. Without it, a photographed receipt is stored and shown normally - only its text is not searchable.

The application shells out in exactly four places, and these five tools are all of them: `taxexport.j` merges with pdfcpu or qpdf, `receipts.j` extracts text with pdftotext and ocrmypdf, and `backup.j` runs mariadb-dump and tar. A tool the code cannot name is a tool the image should not carry - `pdftk` was installed here until it turned out nothing could call it.

If neither merge engine is present, PDF attachments are not merged into the report: a placeholder page names each file and its checksum, and the ZIP carries the originals. Evidence is never silently left out.

Configuration

Everything comes from the environment; `.env` fills in what the environment does not set, and a real environment variable always wins.

Setting	Meaning
APP_DB_DSN	MariaDB DSN. No parseTime=true
APP_DB_WAIT_SECONDS	how long to wait for the database at startup (default 60, 0 = one attempt)
APP_DB_AUTOMIGRATION	apply pending migrations at startup (default false)
APP_AUTH_MODE	dev, authelia or oidc
APP_OIDC_*	the OpenID Connect client, in oidc mode; see Authentication
APP_TRUSTED_PROXIES	required in authelia mode; unused in oidc mode
APP_GROUP_ACCESS / APP_GROUP_ADMIN	the two groups read from Authelia
APP_SECRET	signs cookies and CSRF tokens; required in authelia mode
APP_BASE_CURRENCY	default EUR
APP_COVERAGE_THRESHOLD_PERCENT	family-allowance threshold, default 50

APP_RECEIPT_DIR / APP_EXPORT_DIR	on the app-var volume in compose
APP_DOCS_DIR	the rendered handbook, served at /docs; /app/book-output in the image
APP_FLOAT_MINIMUM	the smallest credit the application spends by itself (default 1,00)
APP_BACKUP_*	the nightly backup; see below
APP_SMTP_* / APP_MAIL_*	outgoing notification mail; see below
APP_MAX_UPLOAD_BYTES	largest accepted attachment; 10485760 is also the ceiling, see below
APP_PDFCPU_BIN / APP_QPDF_BIN / APP_PDFTOTEXT_BIN / APP_OCRMYPDF_BIN	external binaries; empty disables one

Starting without an orchestrator that orders things

`docker compose up` runs the database first, waits for its health check, runs the migrations to completion and only then starts the application. **Swarm ignores depends_on, and Kubernetes never had it** : there all three start together, and the application regularly wins the race against a database that still has a data directory to check.

Two things make that survivable, and both are on by default:

- **The application waits for the database.** `APP_DB_WAIT_SECONDS` (60) bounds it, one line goes to the log while it waits, and it connects the moment the server answers. Only a server that is not listening yet is waited for - "Access denied" or an unknown database fails immediately, because that answer will be the same in a minute.
- **It says when the schema is not there.** A start against an unmigrated database logs `SCHEMA: N migration(s) are not applied yet - run \` fk migrate up\ instead of failing one page at a time with "table expenses does not exist".

Migrations are a job somebody has to run - or the server can run them itself.

By hand , which is the default and what a cautious deployment does: apply them before the new version starts, with a backup already taken.

```
docker run --rm --env-file swarm.env $FK_IMAGE run bin/fk migrate up
```

APP_DB_AUTOMIGRATION=true , for a deployment that would rather not have a separate step: the server applies what is pending before it serves anything, and says so.

```
SCHEMA: 10 migration(s) applied (APP_DB_AUTOMIGRATION)
```

Several replicas starting together are safe: the migration is taken under a MariaDB named lock (`GET_LOCK`), so the second task waits for the first instead of applying the same change beside it, and a task that dies mid-migration releases the lock when its connection closes rather than blocking the next start for ever. If the lock cannot be had within a minute, that server refuses to start rather than serving against a half-migrated schema.

What it does not do is take a backup, and it cannot roll a migration back. That is the argument for leaving it off on anything whose data would be missed.

When something crashes

A failing request does not take the server down. `web.j` runs every handler inside a `try` : an uncaught error is written to stderr as `web: unhandled handler error:` … , the request is answered `500` , and the process carries on. That is deliberate on the framework's side - letting it propagate would make any one request a denial of service - and it means a bug in one page costs that page, not the installation.

A process that does exit is restarted by the orchestrator, not by the application. There is no supervisor inside the container, and there should not be: one process per container is what makes the restart somebody's

job.

- **Compose** - `app` carries `restart: unless-stopped` ; `migrate` is `no` (a one-shot), `backup` and `mail` are `on-failure` .
- **Swarm** - `restart:` is **ignored** . `deploy.restart_policy` applies, and its default (`condition: any` , `delay: 5s` , unlimited attempts) already restarts a failed task. Set it explicitly if you want anything else.

What neither of those notices is a process that is alive but wedged. The image therefore carries a health check:

```
HEALTHCHECK --interval=30s --timeout=5s --start-period=20s --retries=3 \
    CMD ["/usr/bin/jennifer", "run", "bin/healthcheck.j"]
```

`bin/healthcheck.j` asks the server's own **/healthz** on the loopback and exits 0 or 1. Swarm replaces a task whose health check fails; plain Docker only marks it unhealthy, which a monitoring system can pick up.

`/healthz` answers `ok` without a session, without rendering and **without touching the database** . That is the point: a health check that queried the database would turn one slow database into every application task being killed and restarted at the same moment. Whether the database is reachable is what `fk doctor` answers, and what every real page answers by itself.

One file, stored once

An attachment is identified by the **SHA-512** of its contents. When a family attaches the same document twice - one invoice covering two children, on both of their expenses - the rows stay separate and the file underneath is shared. Deleting one row leaves the file alone until the last row using it goes.

Sharing is **within a household** . Two families holding the same document is a coincidence, and letting one family's deletion reach the other's storage is not a coincidence worth arranging.

For an installation that predates this, `fk receipt-rehash` reads every attachment once, records what it is, points the later rows at the first copy and removes what nothing references:

```
reading every attachment; this touches each file once
12 hashed, 3 now sharing a file, 46,1 MB freed
```

It is safe to run twice - the second pass has nothing to do - and it is a command rather than a startup step because it reads every file a family owns.

Photographs are stored twice

A receipt photographed at full resolution is tens of megabytes. Sending that to a phone every time somebody opens the entry is what makes the page unusable, so an upload that is a JPEG or a PNG gets a **display copy** beside it - scaled to `APP_DISPLAY_MAX_EDGE` (2400 px) on the long side, quality 82, metadata stripped. Measured on a real 6000x4500 photograph: **8.58 MB original, 760 KB copy** .

The original is never touched. It is the evidence a tax office would be shown, and it stays byte for byte as it left the camera; the export bundle and the ZIP still carry it. The listing offers both - *Anzeigen* opens the copy, *Original* the file itself - because a page that quietly showed something other than what was uploaded would be lying about evidence.

Setting		Default
APP_IMAGEMAGICK_BIN	the tool; empty serves originals only	magick
APP_DISPLAY_MAX_EDGE	longest edge of the copy, in pixels	2400

It is best-effort, like the text extraction: a missing tool, a failure, or a picture already small enough leaves no copy, and the original is served instead. Being unable to shrink a photograph is not a reason to refuse it.

Two details worth knowing. The decode is bounded with `-define jpeg:size=` , so a 75-megapixel photograph never exists in memory at full size - without it ImageMagick allocates around 300 MB, and several uploads at once would end a small machine. And a copy that comes out no smaller than the original is thrown away rather than stored, because it would double the storage for nothing.

Request limits and what they cost in memory

The server is started with `httpd.listenWith` , not `web.run` - `web.run` takes the engine's own defaults, and its default request body is **10 MiB** , which a full-resolution photograph from a recent phone exceeds. That failed as a bare `413` before any handler ran, so it could not even be reported in German.

Setting		Default
APP_MAX_BODY_BYTES	largest request body read at all	52428800 (50 MB)
APP_MAX_INFLIGHT	requests handled at once	40
APP_MAX_UPLOAD_BYTES	largest attachment a person may upload	51380224 (49 MB)

The first two multiply. A request body is read into memory, so the worst case this process can be asked to hold is `APP_MAX_INFLIGHT` x `APP_MAX_BODY_BYTES` - 40 x 50 MB is about **2 GB of RSS** . Lower either one on a small machine, and size the container's memory reservation for it. The server says so at startup:

```
Limits: upload 49,0 MB, request body 50,0 MB, 40 at once (worst case 2,0 GB of memory)
```

The upload limit stays about a megabyte under the body limit , because the file arrives inside a multipart envelope - boundaries, a `Content-Disposition` per part, the other fields of the form. A configuration that lets them meet is refused at startup: otherwise a file that passes our own check is refused by the transport one byte later, as a 413 rather than as the German sentence the limit has.

Above `APP_MAX_BODY_BYTES` the answer is still a bare `413` with no page around it - the handler never runs, so there is nothing to catch. A reverse proxy can return something friendlier (`client_max_body_size` in nginx), and the upload form states the limit so it should not be reached by surprise.

Behind a reverse proxy

The application does not terminate TLS and does not authenticate. Put it behind a proxy that does both, and give it the proxy's address:

```
APP_AUTH_MODE=authelia
APP_TRUSTED_PROXIES=10.0.0.1,172.16.0.0/12
APP_AUTHELIA_URL=https://sso.example.com/
APP_SECRET=<a long random string>
```

The proxy must set `Remote-User` , `Remote-Email` , `Remote-Name` and `Remote-Groups` , and must **strip those headers from incoming requests** so a client cannot supply them itself. The trusted-proxy check is the second line of defence, not the first.

First run

```
docker compose run --rm cli user-add root --name "Administration" --admin
docker compose run --rm cli household-add "Familie Muster"
docker compose run --rm cli member-add martin --role master --name "Martin Muster"
docker compose run --rm cli member-add anna   --role child  --name "Anna Muster"
docker compose run --rm cli doctor
```

The administration account belongs to no family. From **Verwaltung** in the web interface it can create further accounts, lock them, grant or withdraw administration, and act as any of them to reproduce a problem - see Authentication for what that does and does not carry.

`doctor` is the first thing to run when something is wrong: it reports the configuration, whether the connection is strict, pending migrations, whether each directory is genuinely writable, and each external tool's real version.

Directories, at startup

Before it serves anything, the application checks every directory it writes into - `APP_RECEIPT_DIR` ,

APP_EXPORT_DIR , and APP_BACKUP_DIR when backups are armed - and says on the log what it found:

```
DIR: created receipts at /app/var/receipts
DIR: fixed the permissions of exports at /app/var/exports
DIR: backups at /backup is NOT writable (the directory is not writable) -
     on the host, try `chown -R 10001:10001 /backup`
```

Missing directories are created. A directory that exists but refuses a write has its permissions widened to `rwxrwxr-x` and is tried again - which fixes the usual case of a bind mount made by the wrong user, and cannot fix a directory owned by somebody else. Writability is decided by **writing a file** , not by reading the mode bits: under a mapped user id, an ACL or a read-only mount those two answers differ, and only one of them is the one that matters.

A directory that cannot be fixed is a **warning, not a refusal to start** . The pages that do not touch it work perfectly well, and an installation whose exports are misconfigured is better off running and saying so than crash-looping inside an orchestrator. `fk doctor` reports the same three.

State to back up

- the **database** - everything except the files
- **APP_RECEIPT_DIR** - the attachments; the database only holds their paths and checksums

APP_EXPORT_DIR holds generated bundles and can be regenerated.

Mail

Nothing is sent until a relay is named. With APP_SMTP_HOST empty the application queues nothing, the `mail` service starts, says so and exits, and the *Einstellungen* page tells people that no post can arrive - which is better than a switch that quietly does nothing.

Variable	
APP_SMTP_HOST	the relay; empty means no mail at all
APP_SMTP_PORT	default 587
APP_SMTP_SECURITY	starttls (587), tls (465, implicit) or none
APP_SMTP_USER / APP_SMTP_PASS	SASL credentials; the mechanism is negotiated from the relay's EHLO
APP_MAIL_FROM	the envelope sender; required as soon as a host is set
APP_MAIL_FROM_NAME	the display name in front of it (default Familienkonto)
APP_MAIL_SCHEDULE	how often the queue is drained (default */5 * * * *)
APP_MAIL_DUMP_DIR	write messages here as .eml files and send nothing

A request never talks to a mail server. An event writes a row into `mail_queue` in the same transaction as the action that caused it, and the `mail` service takes those rows to the relay. A settlement that is booked has been booked, whether or not anybody's SMTP server was answering; a relay that is down delays post and loses no bookkeeping.

A message the relay refuses goes back in the queue with the reason and is tried again, five times, and is then left as `failed` - never deleted, and never retried for ever:

```
docker compose run --rm cli mail status          # pending / sent / failed
docker compose run --rm cli mail send            # drain it now
docker compose run --rm cli mail retry           # put the failed ones back
docker compose run --rm cli mail test --to me@example.com
```

APP_MAIL_DUMP_DIR is the way to see exactly what would go out before any of it does: set it, let something happen, and read the files. It is also what the tests use, so they need no mail server.

APP_BASE_URL matters here: it is what the links in the messages point at, so it has to be the address people

actually type, not the container's port.

Backups

The `backup` service writes **one archive a day** into `/backup` : a `mariadb-dump` of the database and a copy of the receipt directory, in a single `familienkonto-YYYYMMDD-HHMMSS.tar.gz` . Both halves are needed - the rows point at files and the files mean nothing without the rows - and both are stored in formats that restore without this application:

```
tar -xzf familienkonto-20260830-031500.tar.gz
mariadb -u expenses -p expenses < database.sql
cp -a receipts/. /path/to/APP_RECEIPT_DIR/
```

It is **off by default** . Arm it in `.env` :

Variable	
APP_BACKUP_ENABLED	true to run at all; without it the container starts, says so and exits
APP_BACKUP_HOST_DIR	the host directory bind-mounted onto /backup (default ./backups)
APP_BACKUP_DIR	where the container writes (default /backup)
APP_BACKUP_KEEP	how many archives to keep; 0 keeps every one (default 14)
APP_BACKUP_SCHEDULE	five-field cron expression in the container's TZ (default 30 3 * * *)

The directory has to be **writable by the container's user** (uid 10001), and a directory you create yourself is not:

```
mkdir -p ./backups
sudo chown 10001:10001 ./backups
```

Without that the first run stops with *"the backup directory /backup is not writable"* rather than a `mkdir: permission denied` naming a temporary path nobody has heard of.

`/backup` is a **bind mount, not a named volume** : a backup that lives in the same Docker installation as the database it protects has solved nothing. Point `APP_BACKUP_HOST_DIR` at something that gets copied off the machine.

There is no cron daemon in the image. `fk backup --schedule` is the loop - it computes the next fire with the `cron` module, sleeps in half-minute slices so `docker stop` is noticed, and lets compose restart it if it dies. A failed run logs and waits for tomorrow rather than ending the schedule.

The same command without `--schedule` takes one backup now, which is what to run before an upgrade:

```
docker compose run --rm cli backup
```

`fk doctor` reports the setting, the directory and the newest archive, because a backup nobody has looked at since it was configured is the usual way of having no backup.

The password is never on a command line - `ps` is readable by anyone on the host - but in a `--defaults-extra-file` written `0600` and deleted after the dump.

http and https

Session and CSRF cookies are marked `Secure` when `APP_BASE_URL` starts with `https://` , and not otherwise. That is not a preference: a `Secure` cookie is never sent back over plain http, so an installation served over http with the flag set cannot complete a single form - including the sign-in - while looking perfectly healthy.

So run it behind TLS and set `APP_BASE_URL` accordingly; a plain-http installation still works, with the cookies unprotected against a network attacker, which is the honest trade for a laptop or a demo.

Downloads and umlauts

`Content-Disposition` is an ASCII header, so a receipt called `Rechnung Müller.pdf` is named twice: once transliterated (`Rechnung Mueller.pdf`) for clients that read only the plain form, and once percent-encoded (`filename*=UTF-8''…`) for everything since RFC 6266. Browsers that understand the second prefer it; the first is what the others get, and neither carries a byte the header cannot hold.

Command-line tool

```
bin/fk <command> [options]
docker compose run --rm cli <command> [options]
```

The same modules and the same rules as the web interface, driven from a terminal. It exists for the jobs a browser is bad at - migrations, seeding, producing a tax bundle from a cron job, importing a bank file - and for seeing what the application thinks without clicking through it.

`bin/fk` is the program itself, with `#!/usr/bin/env -S jennifer run` on the first line and the executable bit set. It works out where the checkout is from its own path - `fs.realpath(os.ARGS[0])` , so a symlink is followed - and reads `.env` and writes `var/receipts` and `var/exports` relative to that, not to wherever you are standing. Link it onto `PATH` and `fk report` works from any directory. `jennifer run bin/fk` does the same thing for anyone who prefers to say so.

`--household` is only needed when more than one family **has members** ; a normal single-family installation never needs it.

Payments and their proof

```
bin/fk settle --from martin --to anna --amount 45,00 --method cash
bin/fk receipt 0 auszug.pdf --payment 12 --kind bank_statement --as martin
```

`--method` is one of `transfer` , `cash` , `revolut` , `card` , `other` ; without it a payment is recorded as a transfer. `--kind` is one of `receipt` , `invoice` , `bank_statement` , `payment_slip` or `other` , and defaults to `receipt` ; `payment_slip` is the proof of the one transfer - the banking app's confirmation, the counter receipt - where `bank_statement` is a whole month of them. The `receipt` command takes an expense number as its first argument, or `0` together with `--payment` to hang the file on a settlement instead - one owner or the other, never both.

Which database

`APP_DB_DSN` decides, and the process environment beats the value in `.env` . That is worth knowing when a terminal disagrees with the browser: a server started with `APP_DB_DSN=...expenses_demo` and a `bin/fk` run in a plain shell are looking at two different databases, and neither is wrong.

`bin/fk doctor` prints the one it is using, without the password:

```
Datenbank
  Adresse            127.0.0.1:3307/expenses_demo
  Verbindung         steht
  sql_mode           STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_ENGINE_
SUBSTITUTION
  strict             ja
  offene Migrationen 0
```

Run with no command at all, `fk` prints its usage and stops.

Consistency

`doctor` ends with a section that checks what the schema cannot. Foreign keys hold the *shape* of the data; these hold its arithmetic, and every one of them is a rule that, broken, shows up as a balance nobody can explain rather than as an error:

```
Consistency
  shares add up to the expense           ok
  shares not settled beyond their value  ok
  payments not spent twice               ok
  allocations stay in one family         ok
  accepted shares have both sides        ok
  categories belong to their family      ok
  payment parties are members            ok
  share parties are members              ok
  dedicated payments name a category     ok
  receipt files present                  ok
  nothing to report
```

A failing check reports how many rows break it and the first few ids — `1 row(s), e.g. #18315` — and stops counting at 50, which it says as `50+` . It reads and changes nothing: what to do about a finding depends on how it got there. `receipt files present` walks the attachment rows and looks for the file on disk; the usual cause of a miss is a restore that brought the database back without `APP_RECEIPT_DIR` , and the first thing it breaks is the Finanzamt bundle.

Commands

Command	Does
doctor	check the installation: database, directories, tools, and the consistency of the books
migrate up|down|status	schema migrations
household-add <name>	create a family, with its categories
household-list	families and their members
user-add <user> [--admin]	create an account that belongs to no family yet
demo-seed --yes	wipe everything and install the demonstration families
member-add <user> --role	add a person, creating the account if new
expense-add	record an expense
expense-list	list expenses
pending	what is waiting for approval
review <nr> --as <user>	accept, or --reject to reject
receipt <nr> <file> --as	attach a document
income-add / income-list	a child's own income
balance	wallets, and what the adults owe each other
settle --from --to	record a payment; without --amount it settles everything open, --purpose <code> dedicates it to one category
report --child --year	a report; --tax restricts it to tax-relevant entries
export --child --year	build the PDF and ZIP bundle
backup [--schedule]	one archive of the database and the receipts, or one a day
mail [send|status|retry|test]	take the queued notification mails to the relay
receipt-rehash	give every attachment its SHA-512 and store one file once
trail	read a family's audit trail; --purge N removes entries older than N days

Examples

```
# Anna paid for a schoolbook that a parent bears
familienkonto expense-add --child anna --amount 45,20 --merchant Thalia \
    --description "Schulbuch Mathematik" --category schule --bearer martin

# something she paid for herself: no approval needed, still counts for tax
familienkonto expense-add --child anna --amount 15,00 --merchant Kino \
    --description Kinokarte --category freizeit --private

familienkonto pending
familienkonto review 3 --as martin
familienkonto balance

# settle everything open, with a generated payment reference
familienkonto settle --from martin --to anna

# a lump sum in advance
familienkonto settle --from martin --to anna --amount 200,00

# money that is for the rent and nothing else: it waits for a Wohnen expense
# instead of being spent on the next schoolbook
familienkonto settle --from martin --to anna --amount 890,00 --purpose wohnen

familienkonto report --child anna --year 2026 --tax
familienkonto export --child anna --year 2026
```

The bank-CSV import is switched off

`config.IMPORT_ENABLED` is `false` , and both the web routes and the `import` command are registered behind it. With the flag down `/import` is a 404 and `familienkonto import` is an unknown command; nothing dispatches to `importCommand` , `importPage` , `importPreview` or `importCommit` , and nothing links to them.

The code is not deleted and its tests still run - `bankcsv` and `bankimport` are covered as they were. What is unfinished is not the parsing but the question above it: how a family should reconcile a bank export against entries it has already recorded by hand. Turning it back on is one `true` in `src/core/config.j` , one line in `bin/fk` , and one entry in each of that file's two command maps.

Output

Plain text, aligned into columns, and **English throughout** - the web interface is the family's, this is the operator's, and reading a log line, a stack trace and a command's output in one sitting should not mean changing language halfway. The German error catalogue is deliberately not installed here: the domain modules throw English sentences and that is what an operator sees. Amounts and dates keep the format the data is in.

The report draws the same coverage bar the browser and the PDF show:

```
  [####################|####################] 100,0 % (threshold met)
```

Errors say what is wrong rather than that something went wrong:

```
error: Martin Maier is not a child of this family
error: cannot read that amount: fünf
error: Anna Maier may not review expenses
```

A mistyped command line is answered the same way, with the usage of the command that was typed rather than the interpreter's own error struct:

```
$ bin/fk expense-add
error: missing required flag: --child

Usage: expense-add [options]
...
```

Role checks are enforced here too - `review --as anna` is refused, not merely discouraged.

The first administrator

An empty installation has no families, and `member-add` needs one. `user-add` is the way in:

```
bin/fk user-add root --name "Administration" --admin
```

That account belongs to no family - which is the point, an administrator runs the installation rather than living in it - and can create the rest through **Verwaltung** in the web interface.

Demonstration data

`demo-seed` empties the database and installs two families - the ones the German guides are written around. It refuses unless `APP_AUTH_MODE=dev` , and `--yes` is required, because it deletes everything first.

```
bash scripts/demo-seed.sh          # into a database of its own
```

The data lives in `src/app/demodata.j` , so this command, the button on the development login page and the screenshots in `docs/images/` all install exactly the same thing.

Development and testing

Running from source

```
docker compose up -d db                 # database only
jennifer serve web/app.j                # the web interface
bin/fk doctor                           # the CLI
```

`config.load(".env")` reads `.env` for anything the environment does not set.

Tests

Co-located `MODULE_test.j` overlays, run by `jennifer test` . The overlay has **white-box access** : it sees the module's private functions without importing it, so a helper can be tested directly.

```
export APP_TEST_DB_DSN='expenses:expenses@tcp(127.0.0.1:3307)/expenses_
test?charset=utf8mb4'
jennifer test src/core/ledger_test.j
for t in src/*/*_test.j tests/*_test.j; do jennifer test "$t"; done
```

`APP_TEST_DB_DSN` must point at a **separate** database: `testdb.fresh()` wipes every table between tests. The database-backed tests run against real MariaDB - the schema, the foreign keys and the ordering are what production uses, and a test that passes against a stub but fails against the server is worth nothing.

`tests/` holds the black-box tests, which drive a real server over HTTP rather than calling into a module. `web/app.j` calls `web.run` at the top level, so it cannot be imported by a test - importing it would start the server and never return - and a `web/app_test.j` overlay is therefore impossible. Those tests spawn `jennifer serve web/app.j` as a subprocess instead, so they need `jennifer` on `PATH` and a free port (18099, or `APP_TEST_HTTP_PORT`).

The pure modules need no database at all: `money` , `dates` , `currency` , `ledger` , `coverage` , `roles` , `bankcsv` and `config` are testable on a machine with nothing running.

House style

```
jennifer fmt -w $(ls src/*/*.j web/*.j tests/*.j | grep -v demodata) bin/fk
jennifer lint src/*/*.j web/*.j tests/*.j bin/fk
```

Both must be clean - with **one exclusion** , `src/app/demodata.j` . Its demonstration rows are one line each, a table of what the two families bought, and that reads better than the same rows unfolded into a stack of one-argument-per-line calls. The line-length check is switched off in that file's header (`# lint-disable-file: L203`); `fmt` has no equivalent switch and its hundred columns are not configurable, so it is kept off the file instead. `fmt` reflows aggressively, which is worth knowing before scripting an edit against a source file.

Every exported function carries a docblock with `@param` , `@return` and `@throws` . Comments explain *why* , not what - the code already says what.

Jennifer traps this codebase hit

Worth knowing before writing more:

- **Value semantics.** Passing a list to a helper copies it; the helper cannot append to the caller's. Return the result instead. This produced two silent bugs here, both caught by tests.
- **1..0 is an error** , not an empty range. Guard a loop over a possibly empty list.
- **bytes has no literal.** `def b as bytes;` then `$b[] = 65;` , or `convert.bytesFromString` .
- **Cooked strings interpolate {...}.** Regex quantifiers like `{1,3}` must live in a raw `'...'` string.
- **Identifiers are <= 64 characters** , letters and digits only. Long test names hit this.
- **list is a type keyword** and cannot name a function.
- **The HTTP request body can be read once.** See Authentication.

Adding a feature

1. Put the rule in a **pure module** if it is arithmetic or a decision, and test it without a database.
2. Put persistence in a **repository** , scoped by `household_id` in the lookup rather than checked afterwards - an id from another family should simply not resolve.
3. Add the capability to `roles.j` as an **explicit set** , never derived from an ordering.
4. Wire it into `web/` and `bin/fk` . If the two could disagree, the rule is in the wrong layer.
5. Say so in `docs/` . A capability that only the code knows about is a capability nobody uses.

Messages

The code raises its errors in **English** ; a family reads them in **German** . `src/messages.j` is what sits between: the English source string is the key, the German sentence is the translation, and `intl.tr` falls back to the key, so a message nobody has translated yet shows up in English rather than vanishing.

```
fail("the payment amount must be greater than zero");
failp("no such expense: %id%", {"id": convert.toString($id)});
```

Every module has `fail` and, where a message carries values, `failp` . Both go through the catalogue, so a call site never spells out German and never concatenates a value into a sentence - the `%name%` slots are filled after the translation is chosen, which is the only order that works when the German word order differs from the English.

`messages.install()` runs from both entry points before anything can throw; `tr` and `trp` also install on first use, so a unit test that calls a module directly still gets German.

`src/messages_test.j` reads `src/*.j` , collects the literal behind every `fail` / `failp` , and fails if the catalogue has no entry for it. That is what keeps a message added next year from reaching a family in English. It also checks that every `%slot%` in an English key survives into the German, because a dropped slot loses the one value the reader needed.

What stays English: whatever MariaDB and the drivers say for themselves. Those messages come from the server, land in the log, and are documented in English here.

Schema changes

Add a migration to `src/migrations.j` with a working `down` . Never edit an applied migration - the initial schema was edited freely during development only because nothing was deployed.

Documentation

`docs/` is the book. It is plain Markdown, so it reads on GitHub as it is, and Grimoire (https://grimoire.jennifer-lang.dev/) renders it into a site plus a single PDF.

```
scripts/books.sh                        # site + both PDFs into book-output/
grimoire build                          # the site alone
grimoire serve --watch                  # http://127.0.0.1:8080, reloads on save
```

Two PDFs, one site. The site carries every page in one navigation, because a reader in a browser can go where they like. A PDF is printed, mailed to a tax adviser or read on a train, and there a parent has no use for the data model and an administrator none for *"wie du eine Ausgabe erfasst"* :

File	Contains
familienkonto-handbuch.pdf	the German pages: welcome, children, parents, tax adviser, administrators
familienkonto-technical.pdf	the English pages: this section

`grimoire.toml` therefore sets `[pdf] enabled = false` and the two books have configurations of their own (`grimoire-user.toml` , `grimoire-technical.toml`). Grimoire takes its outline from `SUMMARY.md` in the source directory and a directory can hold only one of those, so `scripts/books.sh` stages each book - a copy

of its pages plus a `SUMMARY.md` of its own under `var/book-*` - and runs `grimoire pdf` against it. Both are rendered from the same Markdown as the site, so no two of the three can describe a different application. Three pages link to it - `introduction.md` , `user-kinder.md` and `user-eltern.md` - with a plain relative link, because those three sit at the top of `src` and the PDF lands at the root of the output directory. A page one level down, such as anything under `technical/` , would need `../` .

The handbook the application serves

The built site is not only for the web: the image copies `book-output/` to `/app/book-output/` , the application serves it at **/docs** , and every page links to it under *Handbuch* . An installation therefore carries the manual for the version it is actually running, instead of pointing at a website that may have moved on.

Two consequences:

- **scripts/books.sh runs before docker build.** The `COPY` fails otherwise, which is the intended order of complaint: at build time, not in a running container whose *Handbuch* link is dead.
- **APP_DOCS_DIR** names the directory (`book-output` by default, and `/app/book-output` in compose). Set it empty and the link answers with a page saying no handbook is installed - which is also what a checkout that has never built the book gets, rather than a bare 404.

The handbook needs no sign-in. Somebody stuck at the login page is exactly the person reaching for the manual, and it says nothing that is not already public. Requests are mapped onto the directory by hand rather than with `web.serveDir` , because that verb maps the whole request path and would look for `book-output/docs/...` ; `tests/handbook_test.j` covers the `../` cases that mapping has to refuse.

Two things to know before editing:

- **A watch rebuild renders the PDF too** , not just the pages. `build.run` reads the same `[pdf] enabled` on every pass, and the PDF is laid out alongside the site rather than after it, so a save costs about as long as the PDF takes and the download stays in step with the page you just changed. It lands a moment after the `rebuilt N pages` line.
- **The PDF is made from these same pages** , so the three "read this as a PDF" callouts appear inside the PDF too, pointing at the file the reader already has open. `[pdf] exclude` drops whole chapters, not paragraphs, so this stays as it is.

`book-output/` is generated and ignored by git. Nothing in it is edited by hand.

Screenshots

The pictures in the German guides come out of a running development server with demo data, so they show the app rather than a mock-up, and the numbers in the prose are the numbers on the screen.

```
bash scripts/demo-seed.sh                       # two families, into expenses_demo
APP_DB_DSN='...expenses_demo...' APP_LISTEN=127.0.0.1:8099 APP_AUTH_MODE=dev \
    jennifer serve web/app.j &
python3 scripts/screenshots.py                  # writes docs/images/*.png
pngquant --quality=65-88 --force --ext .png docs/images/*.png
```

The button labelled *Demodaten wiederherstellen* on the development login page does the same thing through `demodata.install` , which is why the numbers in the prose keep matching the screenshots.

`demo-seed.sh` **drops** its database first, which is why it has one of its own and why the DSN is not the one in `.env` . The seed is deliberately not random: Familie Maier sits above the Familienbeihilfe threshold and Familie Gruber below it, so both states of the coverage bar can be shown. The third outcome - a child whose own income covers every recorded expense, which fails the rule with an empty bar - is not in the seed; `src/core/coverage_test.j` and `tests/coveragepage_test.j` are what pin that one down.

`screenshots.py` fetches each page with a session cookie and renders the saved HTML in headless Chromium. That works because the stylesheet is inline - there is nothing for the browser to fetch - so a file on disk renders exactly as the served page does. Phone screenshots are scaled down afterwards: the site shows

an image at its own pixel width, and a 2x phone capture left alone fills the column and reads as a tablet.

The PDF carries them as of Grimoire 0.3.0, which draws images into the layout. Alt text still earns its keep - it is what a screen reader announces, and what Grimoire falls back to in brackets when it cannot draw a picture - so keep writing it as a sentence that stands on its own.

`[pdf] imageDpi` decides how big they print, and it reads backwards: it is the resolution the pixels are *read* at, so a **higher** number gives a **smaller** picture. At the default 96 every screenshot came out wider than the 17,7 cm text column, was scaled down to it, and filled most of a page on its own - the tall ones ran to 22 cm. At **192** a desktop screenshot is 13,0 cm across and a phone screenshot 6,3 cm, which is about life size for a phone; two figures and their prose now share a page, and the handbook came back from 35 pages to 30.

The column is a ceiling: a picture wider than it is scaled down whatever the dpi says, which is why lowering the number does almost nothing here. Note also that `imageDpi` changes only the printed size - the full-resolution PNG is embedded either way, so it is not the knob for a smaller file. That would mean capturing at a lower scale factor in `scripts/screenshots.py` .