Nothing is ever deleted

Building software that handles other people's money, and other people's mistakes

Munawar Kazmi • munawarkazmi.com

A plain-language guide to **Safina Portal**, a school management system running in production.

Source-available showcase: github.com/munawarkazmi/safina-portal-showcase • Live at portal.safinainternational.com

---

A receptionist, a stack of cash, and a typing error

A parent comes into a school office and pays a month's fees in cash. The person behind the desk records the payment and accidentally types the wrong amount.

They notice ten minutes later. What should the software let them do?

The tempting answer is: let them fix it. Open the record, correct the number, save. The screen now shows the right figure and everyone gets on with their day.

That answer is wrong, and understanding why is most of what this project is about.

Why editing the number is the wrong answer

If the record can simply be changed, then the system can no longer tell you what actually happened. It can only tell you what it currently believes. Those are very different things, and the difference matters enormously as soon as anyone disagrees.

A parent says they paid 5,000 rupees. The system says 3,000. Who is right? If records can be edited in place, there is no way to answer that from the software, because nothing distinguishes an honest correction made ten minutes later from a quiet alteration made three months later. The dispute collapses into one person's word against another's, which in a school handling cash from families is exactly the situation you cannot afford.

So this system does not allow it. Money is never deleted and never edited:

- A payment recorded in error is corrected by recording a **reversal**, which is its own entry with its own timestamp and its own author. Both entries stay forever.
- A charge that should not have been made is **waived**, which again is an action that is recorded, not an erasure.
- The ledger only ever grows.

This is how actual accounting has worked for centuries, and it is how a surprising amount of software does not work. The correction is a fact about the world, and facts do not get deleted because they were inconvenient.

> The principle in one line: the system should be able to tell you what happened, not just what it currently thinks is true.

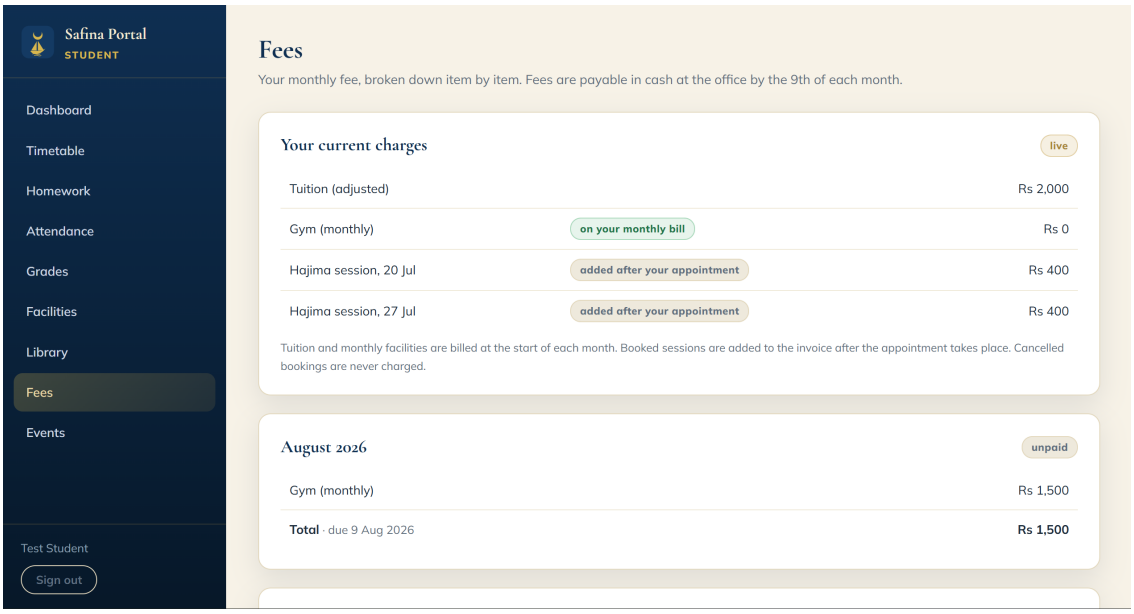

Figure 1: The student's own view of their fees. Note the vocabulary: tuition marked as adjusted, a facility at zero because a special price applies, and sessions added after the appointment took place. The rules above are not hidden in the back office; the person being charged can see exactly how the figure was reached.

Billing that survives contact with a real school

Most billing software assumes a tidy world where everyone starts on the first of the month and pays the same amount. Real institutions are not like that, and the gap between those two things is where administrators end up keeping a parallel set of records in a notebook, which defeats the point of having software at all.

This one is built for the messy version:

- Tuition and monthly subscriptions are billed **in advance**; individual booked sessions are billed **in arrears**, after the appointment has actually happened. Cancelled bookings are never charged.
- Somebody joining on the 17th is charged immediately and **prorated to the day**, not billed for a full month or given a free fortnight.
- Facilities offer a **one-time three-day trial**, followed by a lock so the trial cannot be taken repeatedly.
- Unpaid fees lead to **automatic suspension**, with a grace mechanism, rather than someone remembering to chase it.
- Individual students can be given **special prices, down to zero**, because institutes like this one carry families who cannot pay, and that arrangement should live in the system rather than in someone's memory.

That last point is a design decision rather than a technical one. A family that can see the arithmetic does not have to trust it.

An audit log nobody can go around

Every insertion, change and deletion, across every table in the system, is recorded: who did it, when, and what the values were before and afterwards. It can be filtered by person, by area, and by date.

The important part is not that the log exists. It is where it is written from. The log is produced by triggers inside the database itself, rather than by the application. That means it does not depend on any particular screen remembering to record what it did, and it cannot be bypassed by going around the interface.

Including by the person who built it. That is the point of putting it there.

Everyone sees exactly their own work

Seven kinds of people use this system: students, teachers, administrators, a super administrator, the accounts office, a librarian, and facility staff. Each gets a home screen built for their job rather than a general-purpose dashboard with irrelevant parts greyed out.

Underneath, separation is enforced at the database level rather than by hiding buttons. Every table carries rules about who may read and write which rows, and there are over a hundred such rules across the system. Anything privileged goes through specific guarded operations, and the internal helper functions are withdrawn from the public interface entirely. Photographs and documents live in private storage and are served through links that expire.

The practical consequence: a student who worked out how to send requests directly to the server, bypassing the app completely, would still only be able to read their own records. The interface is a convenience, not the security.

What this guide cannot show you

The other guides in this series end by inviting you to download the code, run one command, and reproduce every number. This one cannot, and the reason is worth stating plainly.

This is a live system holding real families' names, real children's attendance, and real payment records. That data is not publishable and never will be. What is published is the source code, so the design can be read and judged; the data stays private, which is the correct trade even though it makes the work harder to verify.

So this document contains no usage figures, no uptime claims and no performance numbers, because none of those could be substantiated from anything a reader can inspect. What can be checked is what the code does: fifteen ordered database files that grow the system from a prototype to the finished thing, the audit triggers, the billing engine, and the permission rules. Those were read directly rather than taken from the summary at the top of the repository.

One further note: unlike the author's other projects, this repository is source-available rather than open source. It may be read, and it may not be reused or deployed. The institute it serves is a real organisation, and its systems are not a template.

Why the dull parts are the ones that matter

The interesting engineering here is not the screens, which are the part people notice. It is the rules underneath, and almost all of them exist to handle the moment something goes wrong: a mistyped payment, a family that joins halfway through the month, an argument about what was paid in March, a member of staff who should not be able to see somebody's medical notes.

Software that works when everything goes correctly is not difficult. Software that behaves sensibly and traceably when a human makes an ordinary mistake is the whole job, and in an institution where the users are not computer experts and the money is cash, it is the difference between a system that gets adopted and one that quietly gets replaced by a notebook.

Built and operated end to end for Safina International, Multan. This guide is part of a wider set covering the author's public projects, most of which publish their data and let readers re-derive every claim; this one is the exception, and says so.
Repository: github.com/munawarkazmi/safina-portal-showcase