* Code/features are not binary (work/doesn’t work), they are on a spectrum
* Errors are just probabilities waiting to happen
* Improbable things happen a lot
* Rigid code is resilient until it isn’t
* Resilient code is designed to fail

1. Extract a “save” method
2. Ask: **Can I break this?** (How can it theoretically fail?)
3. Refactor for resiliency
4. Repeat

* Use transactions for database operations that depend on one another
* High value actions have side effects
* Side effects are saves

Problem A: The Account Transfer

|  |  |
| --- | --- |
| Ways it can break | What we can do |
| Data is only partially saved when one more more steps fail | Use a transaction to guarantee atomic (all-or-nothing) execution |
| Two processes try to update the same data at the same time | Use a active record lock (optimic or advisory locks) to prevent concurrent data access |
| A transactional operation has side-effects outside of the DB | Use a database-backed queue to safely unblock the critical path |

Problem B: The Purchase Button

|  |  |
| --- | --- |
| Ways it can break | What we can do |
| A save method must wait for the result of an external API request | Split the operation into steps and capture state asynchronously (purchase is in progress, purchase confirmed, etc.) |
| An external request meant to run only once runs more than once | Use idempotency and retries to ensure eventual completion (may be contingent on API service you are using) |