TSQL.APP — Mental Model

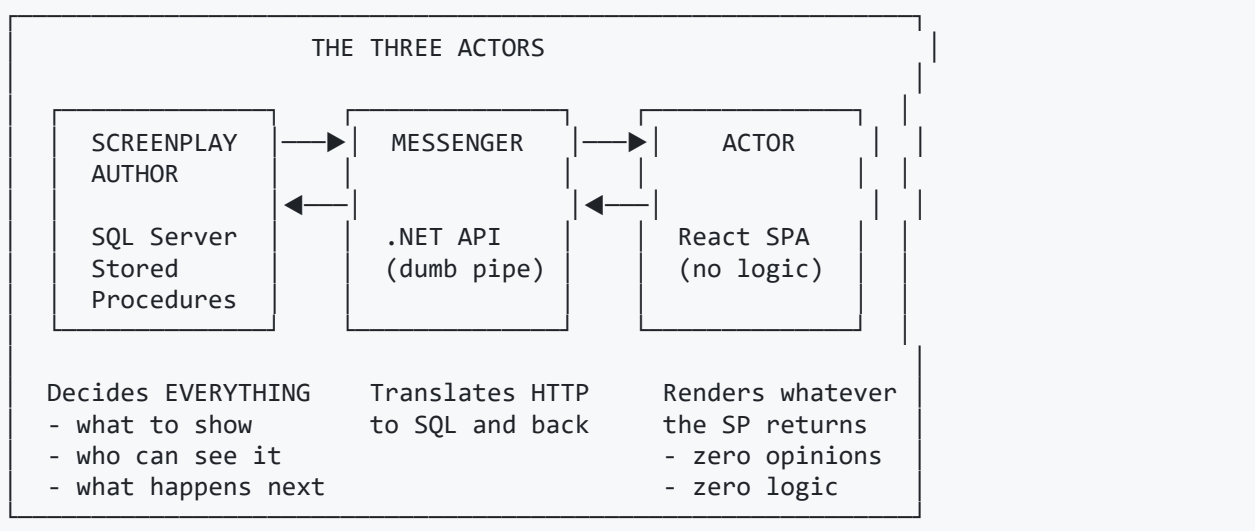
How a SQL Server database becomes a complete application platform.

The Core Insight

TSQL.APP is a framework where SQL Server stored procedures are the application. There is no backend language, no ORM, no service layer. The database contains all business logic, all UI definitions, all navigation, all security — and a thin .NET API + React client simply render whatever it returns.

This is not "database-heavy" architecture. This is **database-only** architecture.

The Three Actors

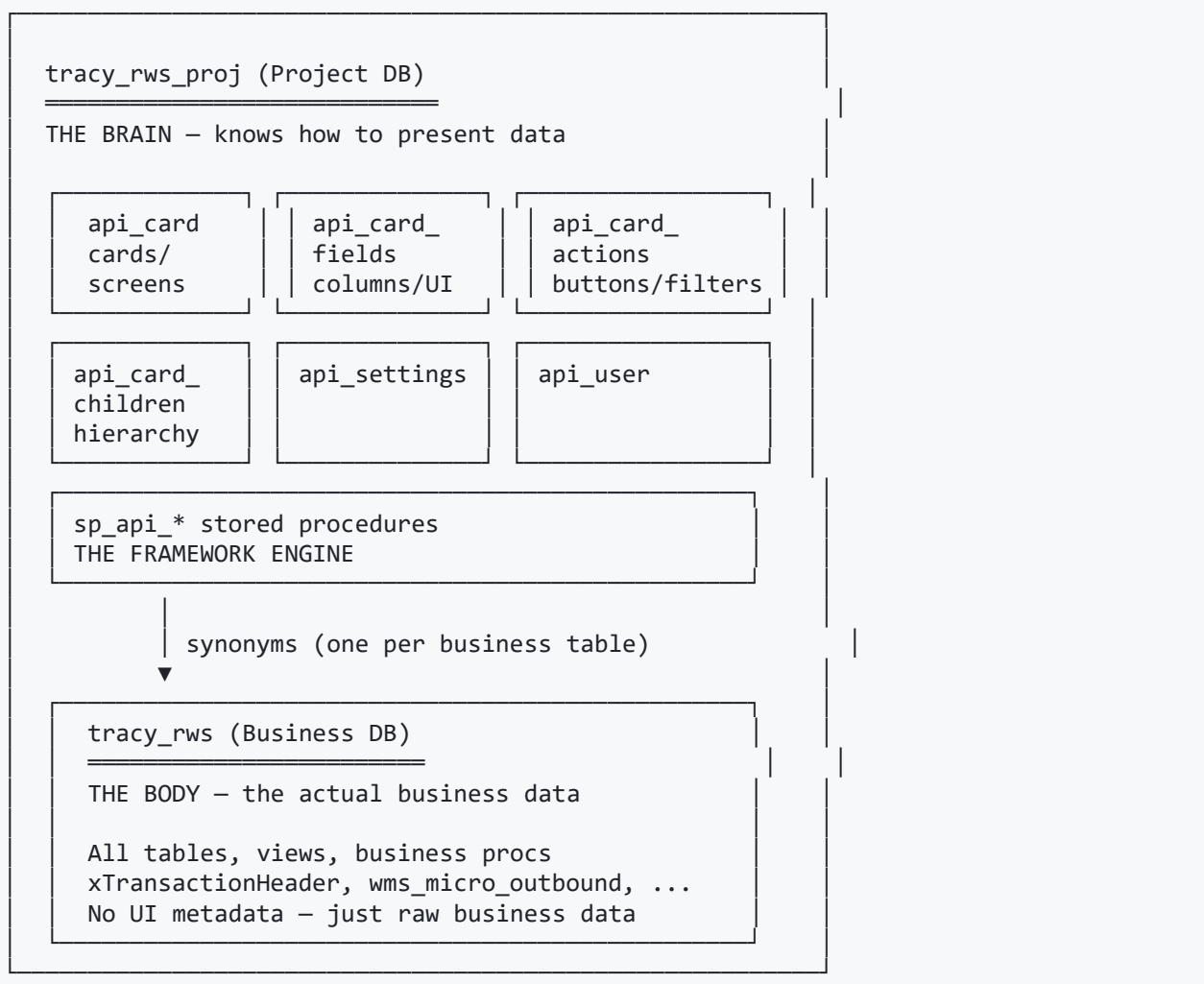


The API has zero business logic. Every HTTP request — GET, POST, PUT, DELETE — is packaged into a context table and handed to one stored procedure. The SP decides everything: what data to show, what UI to render, what buttons are available, what happens when you click them.

The React client has zero opinions. It receives JSON describing components (`Listview`, `Handlerview`, `Modalview`), column definitions, settings, and renders them. It never decides what to show — it only knows *how* to show it.

The Two Databases

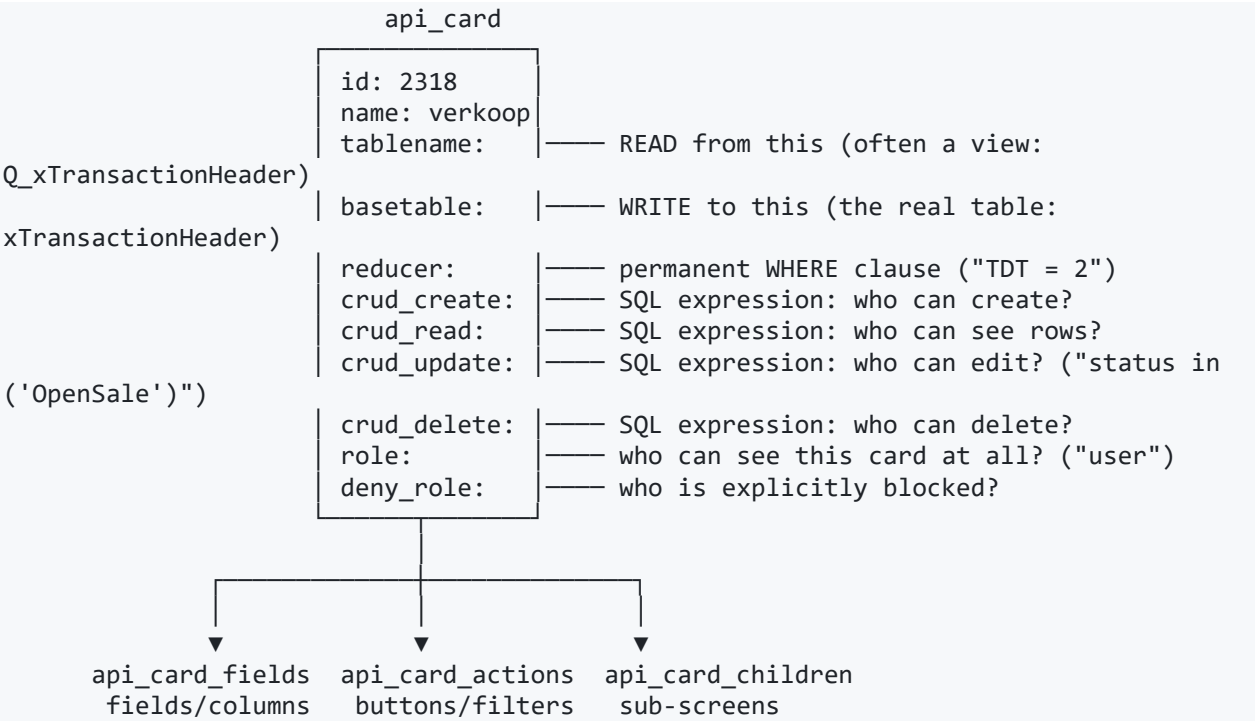
Every TSQL.APP application ("body") is made of two databases:



The project DB connects to the business DB through **synonyms** — one for every business table and view. This lets stored procedures in the project DB query business tables transparently, as if they were local.

The Card: The Universal Building Block

A **card** is the atomic unit of TSQL.APP. Every screen in the application is a card. A card is defined by a single row in `api_card` and connects to:



What makes a card powerful:

1. The tablename/basetable split

- `tablename` = what you SELECT from (often a rich view with JOINs and computed columns)
- `basetable` = what you INSERT/UPDATE/DELETE against (the real underlying table)
- Many cards use a view for reading — the framework handles the split automatically

2. Dynamic CRUD expressions

- CRUD permissions are not static booleans — they are **SQL expressions evaluated per row at runtime**
- `crud_update = "status in ('OpenSale')"` means: you can edit this row only if its status is OpenSale

- `crud_read = "dbo.hasRole('admin,directie') > 0"` means: only admin and directie roles see this data
- The framework injects these as WHERE clause fragments and role flags

3. Reducers as permanent filters

- A card's `reducer` is a WHERE clause permanently applied to all queries
- `verkoop` (sales) and `inkoop` (purchases) share the same table `xTransactionHeader` but use different reducers: `TDT = 2` vs `TDT = 1`
- One table, multiple cards, different views of the same data

---

The Card's Anatomy: Fields

Each card has fields defined in `api_card_fields`. Fields are not just column mappings — they are a full UI specification:

```
api_card_fields
┌──────────────────────────────────────────────────────┐
│                                                      │
│  REGULAR FIELD: sql IS NULL                          │
│    name = "status" → maps directly to table column   │
│                                                      │
│  CALCULATED FIELD: sql IS NOT NULL                   │
│    name = "pallet_count"                             │
│    sql = "(select count(*) from pallets              │
│            where transaction_id = this.id)"          │
│    → computed at query time, 'this' = current row    │
│                                                      │
│  DISPLAY ORDER                                       │
│    list_order    → column position in list view      │
│    detail_order  → field position in form view       │
│    handler_order → position in record preview panel  │
│    insert_order  → position in new-record form       │
│    seek_order    → position in search bar            │
│                                                      │
│  RELATIONSHIPS                                       │
│    picklist_card_id → FK to another card (dropdown)  │
│    picklist_type    → 2=linked picker, 4=JSON list   │
│    picklist_sql     → custom dropdown query          │
│                                                      │
│  BEHAVIOUR                                           │
│    on_update_sql    → fires when field value changes │
│    instant_update   → save immediately on change     │
│                                                      │
└──────────────────────────────────────────────────────┘
```

Field Attributes: The Dynamic Override Layer

On top of fields, there are **field attributes** from `api_card_field_attributes`. Each attribute is a named property with a SQL expression that is evaluated at render time:

Attribute	What it does
`is_updateable`	Dynamic editability: `"iif(dbo.hasRole('admin')>0,1,0)"`
`fill_with`	Default value for new records: `"getdate()"`, subqueries
`element`	UI element override: `"Select"` → renders as dropdown
`is_hidden`	Conditional visibility: `"iif(@status='closed',1,0)"`
`mask`	Input format mask
`extra`	JSON array for dropdown options via SQL query
`is_nullable`	Allow NULL in form
`valid`	Server-side validation expression
`class_name`	CSS styling
`rows`	Textarea height

The key insight: every attribute's `sql` column is evaluated per-record, so a field can be editable for one row and read-only for another, visible in one context and hidden in another — all driven by SQL expressions, no application code required.

The Action System: Buttons That Run T-SQL

Cards have **actions** stored in `api_card_actions`. Actions are not just buttons — they serve three distinct purposes:

```
api_card_actions
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│  action = 'stored_procedure'                                 │
│  ═══════════════════════════                                 │
│  A BUTTON that runs a T-SQL batch script.                    │
│  The sql column IS the entire application logic.             │
│                                                              │
```

```
|  type = 'list'        → button on list view              |
|  type = 'form'        → button on form/detail view       |
|  type = 'list_form'   → button on both                   |
|  type = 'hidden'      → runs via keycode only            |
|                                                          |
|  action = 'reducer'                                      |
|  ═══════════════════                                     |
|  A FILTER. The sql column is a WHERE clause fragment.    |
|  Shows as a filter button in the list toolbar.           |
|                                                          |
|  Example: "id in (select mob_lf                          |
|            from bonded_out_pallets)"                     |
|                                                          |
└──────────────────────────────────────────────────────────┘
```

The Action Script Execution Model

This is the most unusual aspect of TSQL.APP. Action scripts are **re-entrant T-SQL batches** — they run from top to bottom on every user interaction:

```
-- 1. DECLARE all variables
DECLARE @MyInput NVARCHAR(MAX);
DECLARE @MyButton NVARCHAR(MAX);

-- 2. SYNC state (read what the user has typed/clicked so far)
EXEC sp_api_modal_get_value @name=N'@MyInput',  @value=@MyInput  OUT;
EXEC sp_api_modal_get_value @name=N'@MyButton', @value=@MyButton OUT;

-- 3. DRAW the UI
EXEC sp_api_modal_text   @text=N'Enter quantity', @class=N'h3';
EXEC sp_api_modal_input  @name=N'@MyInput', @value=@MyInput OUT;
EXEC sp_api_modal_button @name=N'@MyButton', @value=N'Confirm';

-- 4. PAUSE — if button not clicked yet, stop here
IF @MyButton IS NULL RETURN;

-- 5. VALIDATE
IF TRY_CAST(@MyInput AS INT) IS NULL
BEGIN
    EXEC sp_api_toast @text=N'Must be a number';
    EXEC sp_api_modal_value @name=N'@MyButton', @value=NULL;  -- reset button
    RETURN;
END

-- 6. DO THE WORK
UPDATE inventory SET quantity = @MyInput WHERE id = @id;
EXEC sp_api_modal_clear;
```

The script re-runs from line 1 every time the user interacts. The `sp_api_modal_get_value` calls restore previous state. The `IF @MyButton IS`

`NULL RETURN` acts as a pause — the UI renders up to that point and waits. When the user clicks, the script runs again, this time past the RETURN, and executes the work.

This pattern enables **multi-step wizards, confirmations, and complex workflows** — all in pure T-SQL, with no frontend code.

The Modal Engine: A UI Toolkit in SQL

The `sp_api_modal_*` family is a complete **UI toolkit callable from T-SQL**:

Layout & Text

`sp_api_modal_text`, `sp_api_modal_html`, `sp_api_modal_image`, `sp_api_modal_chart`

Input Controls

`sp_api_modal_input`, `sp_api_modal_input_decimal`, `sp_api_modal_input_date`, `sp_api_modal_input_time`, `sp_api_modal_input_multi`, `sp_api_modal_select`, `sp_api_modal_multi_select`, `sp_api_modal_switch`, `sp_api_modal_choose`, `sp_api_modal_editor`

Buttons & Actions

`sp_api_modal_button`, `sp_api_modal_action`, `sp_api_modal_sure`, `sp_api_modal_button_back`

Tables & Data

`sp_api_modal_table`, `sp_api_modal_table_edit`, `sp_api_modal_table_json`, `sp_api_modal_csv`

Files & Media

`sp_api_modal_file`, `sp_api_modal_file_multi`, `sp_api_modal_file_thumbnails`, `sp_api_modal_download`, `sp_api_modal_audio`, `sp_api_modal_media`, `sp_api_modal_signature`

Device & Hardware

`sp_api_modal_scanner`, `sp_api_modal_qr_scanner`, `sp_api_modal_qr_code`, `sp_api_modal_beep`, `sp_api_modal_screenshot`

Documents

`sp_api_modal_html2pdf`, `sp_api_modal_to_pdf`, `sp_api_modal_pdf_merge`, `sp_api_modal_preview_report`, `sp_api_modal_instant_report`

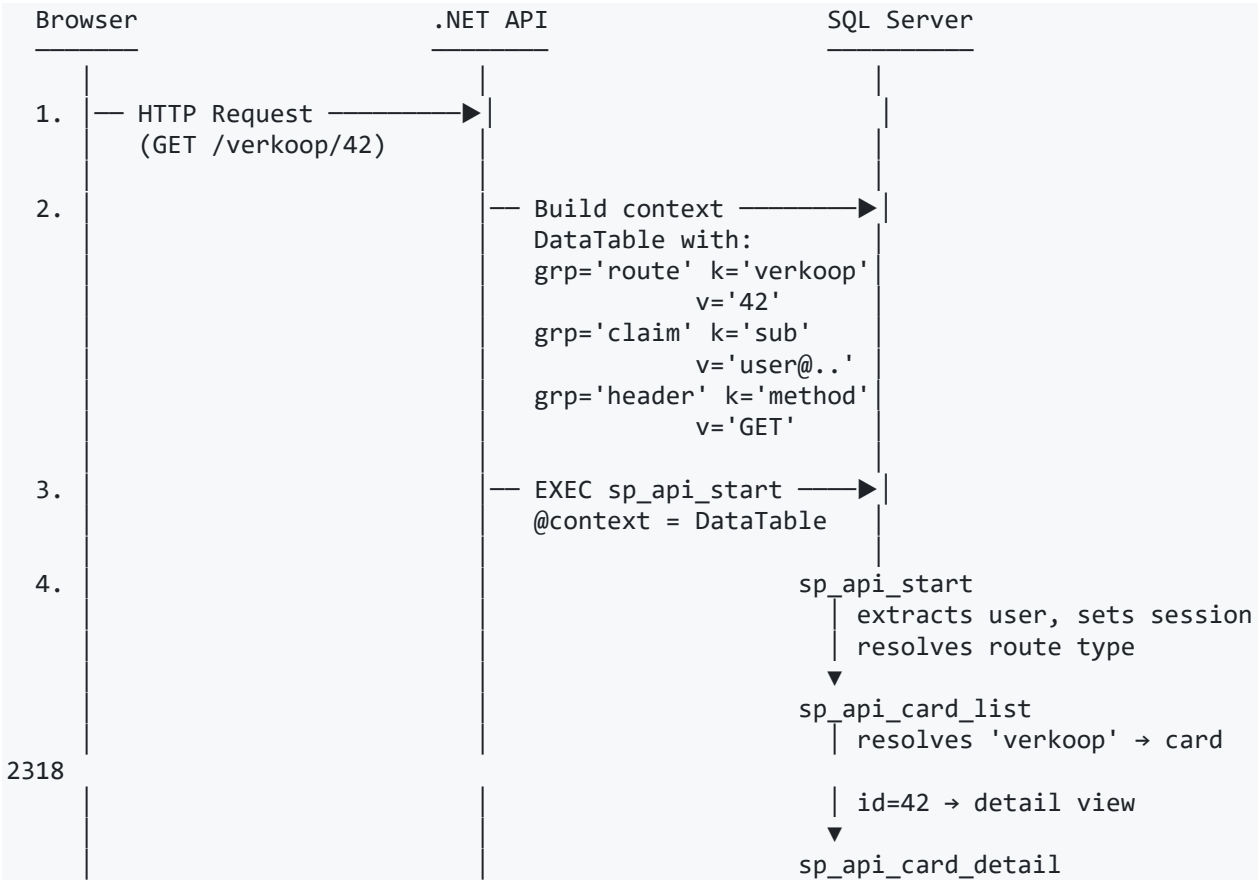
Navigation

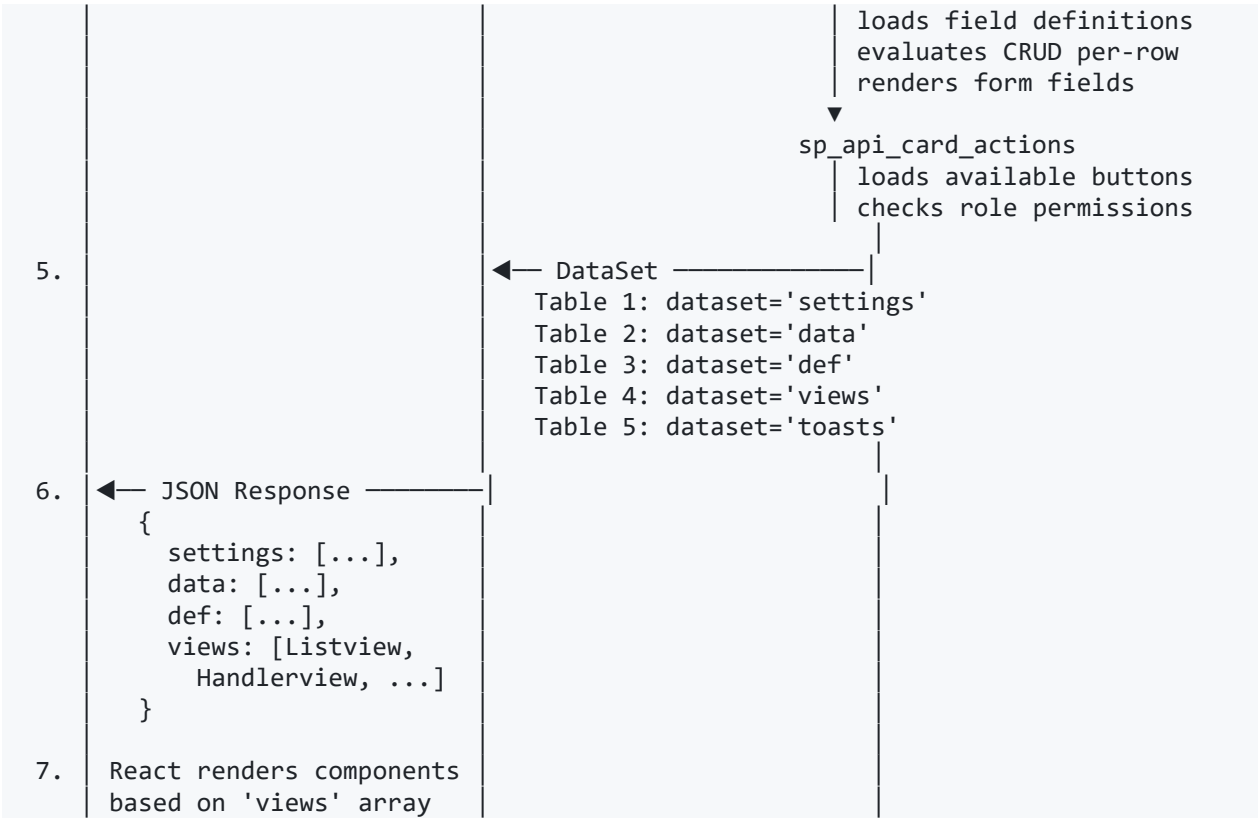
`sp_api_modal_route`, `sp_api_modal_reload`, `sp_api_modal_restart`, `sp_api_modal_dispatch`, `sp_api_modal_focus`

All these write JSON elements to a `#modalview` temp table. The React client renders them as a modal dialog overlay. A developer building a TSQL.APP feature never writes HTML, CSS, or JavaScript — they call stored procedures.

The Request Lifecycle

Every user interaction follows the same path:





The Context Table: The Universal Request Format

Every request is flattened into a single table with four columns:

id	grp	k	v
1	route	verkoop	42
2	claim	sub	user@company.com
3	claim	role	admin
4	claim	role	user
5	header	method	GET
6	header	remote-ip	192.168.1.100
7	header	agfx-pathname	/verkoop/42
8	query	ord	3d
9	cookie	lang	nl
10	version	dotnet	231221

This table IS the request. The SP never sees HTTP, never parses URLs, never reads headers. It reads this table with `SELECT v FROM @context WHERE grp='route' AND k='verkoop'`. The entire web layer is abstracted away.

Navigation: Cards as a Tree

Cards form a **tree** through `api_card_children`:

```
  /verkoop                          ← card: verkoop (sales orders)
  /verkoop/42                       ← record 42 detail view
  /verkoop/42/lines                 ← child card: order lines
  /verkoop/42/lines/7               ← line 7 detail view
  /verkoop/42/lines/7/allocations   ← grandchild: allocations
```

Each segment is resolved by `sp_api_card_list` calling itself recursively. The parent→child link in `api_card_children` specifies:

- **`ref`**: the foreign key column (e.g., `Transaction_ID`) — injected as WHERE clause
- **`reducer`**: additional filter for the child in this parent context
- **`unbound`**: if true, child has no FK — it's a free-standing card shown as a tab

A complex card like `verkoop` (sales) can have many child cards — each one a complete sub-application with its own fields, actions, and even grandchildren.

Security Model

Security is layered and evaluated at every level:

```
LAYER 1: JWT Authentication
  ├── Token validated by .NET API against https://id.tracy.nu
  ├── Claims (sub, role, given_name, ...) injected into context table
  └── SP reads: SELECT v FROM #context WHERE grp='claim'

LAYER 2: Card-Level Access (api_card.role / deny_role)
  ├── tvf_hasRole_v3(role, deny_role) checks user's roles
  ├── Card not accessible → route error
  └── Most cards have role restrictions

LAYER 3: CRUD Expressions (per-row, per-operation)
  ├── crud_create = "dbo.hasRole('admin,directie') > 0"
  ├── crud_update = "status in ('OpenSale')"
  ├── crud_delete = "dbo.hasRole('admin') > 0"
  └── Evaluated as SQL WHERE fragments → controls what the user can do to each row

LAYER 4: Field-Level Editability (api_card_field_attributes)
  ├── is_updateable = "iif(dbo.hasRole('admin')>0,1,0)"
  ├── is_hidden = "iif(@status='closed',1,0)"
  └── Each field independently controlled per-record

LAYER 5: Action Visibility (api_card_actions.role)
```

```
  ├── Each button has its own role requirement
  └── Buttons appear/disappear based on user role
```

All security is enforced server-side in the SP. The React client never makes security decisions — it simply doesn't receive data or buttons it shouldn't show.

The Data Write Paths

There are exactly three ways data gets written:

```
PATH 1: Inline Edit (Tab-to-save)
  User changes a cell in the list → sp_api_card_update_from_body
  Direct write to business table.

PATH 2: Form Edit (Modify/F2)
  User edits fields in form view → sp_api_modal_post (stages to api_storage)
  User clicks Save → sp_api_card_update_from_storage (commits to business table)
  Two-phase: stage, then commit.

PATH 3: Action Script
  User clicks button → sp_api_card_actions_execute
  Script runs arbitrary T-SQL — can INSERT, UPDATE, DELETE anything.
  This is the escape hatch: any business logic that doesn't fit the card model
  goes into an action script.
```

How an Application Gets Built

Building a TSQL.APP application requires zero frontend code. The developer works entirely in SQL Server:

Step 1: Create a business table

```
CREATE TABLE xProduct (id INT IDENTITY, name NVARCHAR(128), price DECIMAL(18,2),
active BIT DEFAULT 1)
```

Step 2: Create a synonym in the project DB

```
-- In tracy_rws_proj:
CREATE SYNONYM xProduct FOR tracy_rws.dbo.xProduct
```

Step 3: Create a card

```
INSERT INTO api_card (name, tablename, basetable, role)
```

```
VALUES ('products', 'xProduct', 'xProduct', 'user')
```

Step 4: Fields auto-populate

`sp_api_card_list` calls `sp_api_populate_card` automatically when it encounters a card with no fields — fields are created from the table's column definitions.

Step 5: Customize

- Set `list_order`, `detail_order` to control which fields appear where
- Add `fill_with` attributes for default values
- Add `is_updateable` attributes for conditional editability
- Add computed fields with `sql` expressions
- Add action buttons with `insert_card_action`
- Add child cards via `api_card_children`
- Add reducers for named filters

Result: a complete CRUD application with list view, form view, search, paging, sorting, filtering, role-based security, and action buttons — without writing a single line of frontend code.

The Embedded AI Layer

TSQL.APP includes two AI assistants that live entirely inside SQL Server:

CODY — The Action Script Developer

- Writes T-SQL action scripts through a chat interface
- Has its own memory architecture across multiple tables for persistent learning, session state, and pinned examples
- Has a dynamic tool registry for schema inspection, code execution, and more
- System prompt is stored in a table and can self-modify
- Code snippets available as `sp_api_snippet_*` procedures

TRACY — The Conversational Orchestrator

- Multi-model orchestration (Grok, Claude) via `sp_api_chatbot_conversation_think`
- Can execute SQL, inspect schemas, send emails, generate documents

- Uses Service Broker for async operations

Both AIs operate **inside the database** — their prompts, memory, tools, and outputs are all SQL Server tables and procedures.

Why This Architecture Works

1. Single source of truth. The database IS the application. There is no impedance mismatch, no ORM mapping layer, no stale cache. When a table changes, the card reflects it immediately.

2. Instant customization. Adding a column to a table makes it available to the card. Adding an action script creates a button. No deployment, no build, no frontend changes.

3. SQL is the logic language. Every developer who knows T-SQL can build features. The modal procedures are the UI toolkit. The card metadata is the layout engine. Complex business logic stays where the data is — in the database.

4. The thin API is replaceable. The .NET API is a dumb pipe — proven by the fact that it was migrated from .NET 2.2 to .NET 9 without changing a single stored procedure. The entire application continued to work because all logic lives in SQL Server.

5. Security by architecture. The client never receives unauthorized data — it can't, because the SP never returns it. There is no client-side route guard or token check that can be bypassed. The database decides, the client obeys.