ChatGPT

TSQL.APP: A SQL-First Platform for Data-Heavy Business Applications

TSQL.APP is a low-code, **database-native application platform** that enables building enterprise web applications *entirely* within Microsoft SQL Server [1]. In other words, it makes SQL Server the central engine for all aspects of an app – data storage, business logic, **UI generation**, state management, and even integrations – using **T-SQL as the sole development language** [2] [3]. This is a fundamentally different approach from conventional web frameworks, and it's designed specifically for **data-heavy business domains** like ERP (Enterprise Resource Planning), warehouse management, and other database-centric systems [4]. Below, we dive into TSQL.APP's architecture and unique capabilities, and how they potentially **redefine business software development**.

Architecture and Development Model

TSQL.APP's architecture centers around SQL Server. There is no separate application server in the traditional sense – **SQL Server itself is the runtime for all application logic** [5]. A minimal web layer (built with .NET Core/Node.js) and a React frontend are used, but these are generic and **stateless**. The React **client acts as a universal UI renderer** that interprets JSON responses from the database to display the UI using Bootstrap styling [6]. This means the client contains no page-specific logic; it's simply a shell that renders what the server tells it to.

One notable design choice is the **two-database architecture** for each application [7]. TSQL.APP splits the data into:

- **Source Database** – the primary business data, which can be an existing ERP or operational database. This remains **unmodified**, preserving all existing tables, schemas, and data [7].
- **Project Database** – the TSQL.APP metadata and logic store. This holds the framework's stored procedures, UI configuration (metadata about forms, cards, etc.), and any custom application code. It connects to the source data via SQL synonyms, so it can query or update the source DB seamlessly [8].

This separation allows organizations to **overlay TSQL.APP on top of legacy databases** without altering them, which is ideal for modernizing systems: you keep your data where it is, and add a TSQL.APP "project" database to handle the new UI and logic. The platform is cross-platform (runs on Windows or Linux SQL Server) and supports scaling features like load balancing and connection pooling for performance [6]. Each tenant or customer can even be deployed on an isolated server (VPS) if needed for security or scaling, according to the platform's deployment model [9].

SQL-First Development Workflow

TSQL.APP adopts a **SQL-first development paradigm**, meaning you build the application by designing the database and writing T-SQL code, and the UI is derived from that. The typical development workflow is: 1. **Database Design** – Define your data structures: create or reuse tables, views, and stored procedures that represent the core business data and logic [10].
2. **UI Configuration** – Define **"Cards"** (which are essentially UI views tied to tables or queries) to automatically generate forms, lists, and detail pages based on the schema [11]. Simply creating a new

table or a table-backed view in the IDE will produce a new Card in the app UI [12].
3. **Logic Implementation** – Implement business rules and interactivity in T-SQL. This could mean writing stored procedures that respond to button clicks, enforce validations, or perform calculations – all within the database. You use T-SQL (and the provided API procedures) to manipulate data and UI state [13].
4. **Instant Deployment** – Deploy and test instantly. Because everything is interpreted by the SQL engine, any change you make (a new table, a modified procedure, etc.) is **immediately live** in the running application with no compilation or rebuild needed [14]. This fosters rapid iteration and prototyping.

All development is done in the browser via TSQL.APP's **integrated IDE**, which is built into the platform [15]. The IDE uses a Monaco editor (the same core as VS Code) for T-SQL editing, providing features like syntax highlighting and autocomplete [16]. Developers never have to leave this web-based IDE – they can create database objects, write procedures, and run/test them all in one place [17]. This self-contained, in-app development experience means faster turnaround and a more unified workflow (**development, testing, and usage all occur in one environment** [18]).

Key Features and Capabilities

TSQL.APP provides a rich set of features that set it apart from traditional frameworks and make it well-suited for heavy data-oriented applications:

Pure T-SQL, Full-Stack Development

One of the most unique aspects of TSQL.APP is that **the entire application stack is implemented in T-SQL**. There's no need to write JavaScript for the frontend or C#/Java for the backend – **stored procedures in SQL Server define everything from business logic to UI behavior** [5] [3]. This drastically simplifies the technology stack: a developer who knows SQL can build a full web application without learning new languages or frameworks. It also means the database is not just for persistence; it becomes the **execution engine for the app**. By replacing the typical web server + API + JS frontend layers with a *single* database-driven environment, TSQL.APP **streamlines the stack** and reduces complexity [19].

Why is this different? In a conventional scenario, you might have to juggle SQL for data, a server-side language for APIs, and JavaScript/HTML/CSS for the UI. TSQL.APP collapses all these layers into T-SQL routines and some JSON metadata. Developers can leverage **existing SQL skills** to handle tasks that would normally require multiple skill sets, resulting in a lower learning curve for those coming from a database background [20]. This "one-language" approach is especially powerful in data-heavy domains, because a huge portion of business logic in ERP-type systems already lives in the database (stored procs, triggers, etc.). TSQL.APP lets that database logic also drive the user interface and workflow, without an impedance mismatch between layers.

Automatic UI Generation with Cards

Building the user interface in TSQL.APP is heavily automated through the concept of **Cards**. A *Card* represents a UI view (such as a form or a data grid) that is directly backed by a database table or view [21]. When you create a new table in the system (using the IDE or via script), the platform will **automatically generate a Card UI for it** [12]. That Card provides a full-featured interface for interacting with the data in that table: it includes standard **CRUD operations**, list/detail views, pagination controls,

search and filtering, and respects role-based permissions – all by default [22]. In essence, the basic front-end for managing a database table is created for you, without writing any HTML or JavaScript.

These autogenerated UIs aren't static; they can be customized and extended. Developers can attach **custom T-SQL code to Cards** to implement special behaviors or triggers on user actions [23]. For example, you might add a piece of T-SQL that runs whenever a record is saved on a particular Card to also send a notification email, or to update related tables. You could integrate calls to external APIs (e.g. to a CRM or shipping service) as part of a button click on a form – all from within the database logic [23]. This ability to inject custom logic means you get the productivity of generated UI **plus** the flexibility of coding business-specific processes when needed.

The UI generated by Cards is also **responsive and consistent** (since it uses Bootstrap under the hood for layout/styling). It supports multiple input types and controls automatically. For instance, if a table column is a foreign key, the Card UI can present it as a dropdown selector. If a field is marked as an image or file, the UI will handle upload/display accordingly. There are even provisions for charts and dashboards – a Card or procedure can generate a chart on the UI using integrated Chart.js visualizations [24]. *In short, TSQL.APP's Card system gives you a working data-driven UI out-of-the-box, which is a huge accelerator for developing data-heavy business apps.* Many ERP systems require dozens of forms and reports; with this platform, those can be generated from schema metadata rather than hand-coded from scratch.

Server-Driven Reactive UI (Roundtrip Model)

Unlike modern single-page applications that keep a lot of state and logic in the browser, TSQL.APP embraces a **server-driven UI paradigm**. Every interaction in the UI (like clicking a button, submitting a form, or even typing into a field) triggers a round-trip to the server where a T-SQL procedure will run and update the interface. The cycle works like this [25] [26]:

- **Initial load:** When a user first opens a page or Card, the framework calls a specific T-SQL procedure (associated with that view) which populates a temporary table `#modalview` with JSON definitions of UI elements (fields, buttons, etc.). That JSON is sent to the client, and the React frontend renders it as a web form or page [27] [28].
- **User interaction:** If the user clicks a button or submits a form, the input data is collected and sent back to the server (in a `#context` temp table as JSON) [29]. Essentially, the frontend posts the current state to the database.
- **Server processing:** The **same T-SQL procedure runs again** on the server, now seeing the `#context` (user input) and perhaps an action flag (which button was pressed). The procedure can branch logic based on that input (e.g., if "Save" was clicked, then perform an update query) [30]. It then recomposes the `#modalview` table to reflect any changes in the UI (e.g., show a success message, or navigate to a different card) [31].
- **UI update:** The server returns a new JSON payload describing the updated UI (or data), which the React client uses to **re-render** the interface accordingly [31].

This tightly looped, **stateless request-response cycle** means the **database is always in control of the application state**. The React front-end is essentially just rendering a snapshot of state given by the server, then sending back user events. There's no complex client-side state management or local business logic to get out of sync. Every user action is handled as a **transactional operation on the server** – ensuring consistency (either the whole operation succeeds and the UI reflects the new state, or it fails and nothing changes). This model also makes the app inherently resilient to connection issues: since each interaction is independent, a dropped connection or a page refresh simply triggers the last known state to be reloaded from the server.

From a developer's perspective, this reactive model is implemented via built-in stored procedures that abstract the details. For example, there's `sp_api_modal_get_value` to retrieve posted form values out of the context, and `sp_api_modal_restart` to push a new UI state to the client [32] [33]. A whole library of `sp_api_modal_*` procedures exists to add UI elements like text, inputs, buttons, toasts, etc., during these roundtrips (more on that below). The key point is that **TSQL.APP makes reactive web apps possible with pure T-SQL** – every event causes a stored proc to run, which can use conditional logic, modify data, and then tell the UI what to do next, all within one consistent environment. This is quite different from traditional web apps where the logic might be split between JavaScript in the client and APIs on the server; here it's **one coherent T-SQL process** driving everything in a loop [34].

JSON-Based State Management

JSON is the lingua franca of TSQL.APP's front-end/back-end communication. All data exchanged between the server and the React client is packaged as JSON payloads. Internally, the platform uses temporary tables (like `#modalview` and `#context`) to hold these JSON structures, and T-SQL code manipulates them using SQL Server's JSON functions. This design has several advantages for state management and integration:

- **Session state as JSON:** TSQL.APP doesn't require writing session-handling code or storing UI state in the database permanently. Instead, state can be kept in the JSON that's passed back and forth on each roundtrip, or in SQL Server's session context if needed. For example, when a form is submitted, all input values come into the procedure as a JSON blob (accessible via `#context` table) [29]. The procedure can parse that JSON (using `OPENJSON` or `JSON_VALUE`) to get the values, perform logic, and then when sending a response, it can include updated values or new UI elements in the `#modalview` JSON. The client will merge that into the interface. This means **multi-step workflows** or wizards can be handled by carrying forward a JSON state between steps, without having to commit partial data to the database [35].

- **Relational + JSON together:** Because SQL Server has robust JSON support (2016+ versions), TSQL.APP leverages that to easily move data between normalized tables and JSON. You can take the result of a complex SQL query and output it as JSON (via `FOR JSON PATH`) to feed into a UI component or to an external API. Conversely, JSON inputs from a web form or external service can be parsed directly into relational form for processing [36]. This dual ability is crucial for modern apps (which often deal with JSON from web services) and TSQL.APP bakes it in at the framework level.

- **Metadata and multi-language support:** The JSON payloads not only carry data, but also metadata that can enable advanced features. For instance, audit information or user-interface hints can be embedded in the JSON sent to the client. One powerful outcome of this is **multi-language UI support** – TSQL.APP can tag UI text with keys and include translated strings for different languages in the metadata. The React renderer then displays the appropriate language to the user. The framework's procedures have parameters to facilitate this; e.g. `sp_api_modal_text` has a `@translate` flag and language context, which if turned on will automatically look up and display the text in the user's preferred language [37]. In other words, you can build a multilingual ERP system by filling a translation table and marking which UI elements should be translated, and the platform handles the rest. JSON makes it easy to pass these translations and context back and forth. (Under the hood, each payload can include labels or messages in multiple languages as needed [38].)

Overall, using JSON as the glue for state and data means TSQL.APP can remain stateless on the web tier and still maintain rich context. This design **borrows the best of web paradigms (JSON, RESTful thinking) but implements them entirely in the database**. It's quite different from an AJAX single-page app that holds a lot of state on the client; here JSON is used to frequently sync state with the server, keeping the source of truth in the database at all times.

Extensive Library of Built-in Procedures

To make T-SQL behave like a full-stack environment, TSQL.APP includes an extensive library of pre-built stored procedures (approximately **400+ procedures and functions** are provided [39] [40]). These cover a wide range of functionality and essentially act as the "framework's API" for developers. Some categories of these procedures include:

- **UI Element Procedures:** A large subset of the library is dedicated to generating UI components in the `#modalview`. For example, `sp_api_modal_text` will add a text or header element (with support for Markdown or HTML, and dynamic translation) [41]. `sp_api_modal_input` creates an input field (text box, date picker, dropdown, etc.) and ties it to a named variable that will be returned upon form submit [42]. There are similar procs for buttons (`sp_api_modal_button`), multi-choice selectors (`sp_api_modal_choose`), pop-up toasts/alerts (`sp_api_toast`), and even for rendering charts (`sp_api_modal_chart`) [43]. The chart procedure, for instance, lets you specify a temp table of data and a chart type (bar, line, pie, etc.), and it generates a Chart.js visualization on the UI – making it trivially easy to add dashboards or reports to your app **without any JavaScript coding** [43]. All these UI procs follow a convention of writing into the `#modalview` JSON; the developer just calls them in sequence, and the framework handles the rest (the client will render them in the order added).

- **Data Operations and CRUD:** Although standard T-SQL can be used for querying and updating the database, TSQL.APP also provides helpers for common patterns. Notably, it can **auto-generate basic CRUD operations** for any table – meaning you get insert, update, delete procedures or UI actions without writing them yourself [44]. The Cards mentioned earlier take advantage of this: a Card will use these generated operations to support editing the table's data through the UI. Of course, you can override or customize these if business rules require it. There are also utilities for things like pagination of query results, or filtering based on user input, to save you from writing boilerplate code.

- **System Integration:** A number of stored procedures enable integrating with external systems or handling tasks beyond basic DB reads/writes. For instance, there are procedures for calling external REST APIs (the platform can perform an HTTP request directly from SQL Server – likely via CLR or an external service it coordinates with) [45]. This means if your ERP needs to fetch the latest exchange rates or send a message to a CRM system, it can be done with a T-SQL call. There's built-in support for messaging via **SQL Server Service Broker** as well, which is useful for asynchronous tasks or communicating between different parts of a system. File management, email sending, and FTP transfers are also supported through provided procedures [45] – common needs in business applications (e.g. generating and emailing an invoice PDF automatically). In short, TSQL.APP's library extends T-SQL's reach to do many things that otherwise would require an application server or additional scripts.

- **Reporting and Documents:** Generating reports or documents is often a big part of ERP/WMS systems (invoices, picking lists, financial reports, etc.). TSQL.APP has **built-in reporting features** to create PDFs, Excel spreadsheets, or HTML reports from your data [46]. Typically, you'd prepare a query or a temp table with the data, then call a procedure to format it or convert it into a

downloadable file. For example, a Card's table view can include an "Export to Excel" button with no coding – just by enabling a flag – because the framework knows how to take the current grid data and produce an Excel file [46]. These capabilities again leverage SQL Server (for data and formatting logic) and the web layer (to deliver the files) without you writing separate reporting software.

- **AI-Assisted Development:** An intriguing modern feature is that TSQL.APP integrates an AI model (custom-trained) to assist with code generation [47]. In the IDE, a developer can get suggestions or even have T-SQL code auto-written by AI based on prompts. This can speed up development for routine tasks or help less-experienced SQL developers by providing templates. It's a feature found in some low-code platforms and IDEs today, and here it's built right into the SQL-first workflow.

The presence of this large procedure library is a key reason TSQL.APP can be so productive. It's *not* just leaving the developer on their own with raw SQL – it provides higher-level abstractions similar to how a traditional framework would provide classes or functions. The difference is, here those abstractions are implemented as stored procs in the database. For example, rather than writing a bunch of code to build an HTML form with validation, a developer calls a handful of `sp_api_modal_input` and related procs in a T-SQL script, and the form appears on the web UI with all necessary behavior. This library approach, combined with the reactive model, essentially turns T-SQL into a scripting language for web app logic. It's a unique **SQL-first runtime** that handles things we don't normally expect SQL to handle (UI, HTTP, files, etc.), blurring the line between database code and application code.

Security, Permissions, and Multi-User Isolation

Security is paramount in business applications, and TSQL.APP's philosophy is to lean on the database for security enforcement. In practice, this means **role-based access control (RBAC) is built-in** and managed at the SQL Server level [48]. You define user roles and permissions within the platform (likely in configuration tables or using SQL Server roles), and the framework can restrict what data or which Cards each role can access. Because all data operations funnel through the database, it's straightforward to ensure *every* action passes the permission checks. This is unlike a typical app where you might need to implement checks in multiple layers (API, UI, etc.). Here, if a user doesn't have access to a certain table or field, the server simply won't include that data in the JSON it sends, or will return an error if an unauthorized action is attempted – all decided in the stored procedures before the UI is built.

TSQL.APP also leverages SQL Server's built-in features for security and reliability: for example, data encryption can be used at rest and in transit (as it's standard in SQL Server), and **audit logging** of all changes can be turned on to comply with regulations (since every change comes through the database, it's easy to log it) [49]. Transactions are used to ensure data integrity – if a multi-step operation fails in the middle, the database can roll it back, and the UI will get an error state without partial data being saved. Essentially, **the database is the trust boundary** [50]. The client is thin and untrusted (it just sends inputs), and all validation happens in T-SQL procedures. This greatly reduces certain security risks, like a malicious user bypassing the UI – they still can't bypass the stored procedures and SQL permissions.

For **multi-user environments**, TSQL.APP ensures that each user's session and data views are isolated from others. Temporary tables like `#modalview` and `#context` are typically scoped per session or connection, so one user's actions won't spill over to another's. Additionally, if multiple users are editing the same data, standard database concurrency control (row locks, transactions) applies. The framework likely has some mechanisms to handle optimistic concurrency or at least will show the latest data when a form is opened (since it always pulls fresh from the DB on each interaction). The platform's docs

mention **session-scoped state and user-tagged data to prevent cross-session leakage** [51]. In practice, that means if two users open the same Card, each has their own copy of the `#modalview` for that Card, and any transient changes or UI state (like a selected dropdown value) is not shared. This is critical in enterprise apps where many users work concurrently on the system – TSQL.APP essentially uses the database's robust session handling to keep everyone separate and secure.

From an administration standpoint, since the entire app is in the database, auditing and monitoring user activity becomes easier too. One can potentially run SQL traces or examine audit tables to see exactly which procedures were run or which records were touched by which user – giving a single point for logging. This consolidated, **database-centric security model** aligns well with enterprises that already trust their DBAs and SQL security for protecting data, and it avoids the complexity of duplicating security rules in multiple layers of an app [52].

Rapid Deployment and Iteration

Because TSQL.APP applications are defined by database metadata and T-SQL code, **deployment is exceptionally quick and simple**. There's no separate application server to build or deploy – once your changes are in the database, they're live. In development mode, as mentioned, you get instant feedback (you edit a procedure and immediately re-run it in the app). For production, rolling out an update could be as simple as running a script to update the database or syncing the new stored procedures to the server. This is a stark contrast to traditional deployments that might involve compiling code, pushing to a web server, dealing with package dependencies, etc. In fact, the platform touts *"bare-metal simplicity"* in the sense that a SQL script is the deployment unit [53].

For businesses, this means the cycle from idea to implementation can be much shorter. Suppose users request a new field on a form and a corresponding report in an ERP system. In TSQL.APP, a developer could add the column to the database table, let the Card auto-update the form UI, and write a small T-SQL snippet for the report – all within a single environment. The update would be immediately visible to testers or users. There's no need to coordinate front-end and back-end teams, because one person can do the change in one place. This agility is a major advantage in rapidly evolving business environments.

Furthermore, TSQL.APP supports versioning and environment management in a streamlined way. Since everything is in the database, creating a new test environment might be as easy as copying a database. The React front-end is generic and doesn't hard-code business logic, so it can point to a different project database (say, a QA database instead of production) to instantly serve a test instance of the app. This **unified deployment model** (one React app + one or more databases) means less overhead in DevOps. Scaling up is also straightforward: if you need to serve more users or more data, you scale the SQL Server instance or add additional read replicas, etc., much like scaling a database – you don't have to scale separate web and app servers independently because the database is doing the bulk of the work.

Impact on Business Software Development

By consolidating application development entirely within the database, TSQL.APP **challenges the conventional approach** to building business software [4]. In a traditional stack for an ERP or similar system, you'd have different layers (UI, API/logic, database) possibly written in different languages by different specialists. TSQL.APP proposes a unified model: **the database engineer effectively becomes**

the full-stack developer, using T-SQL for everything. This can change how teams are structured and how software projects are planned:

- **Shift in Skillsets:** Organizations might rely more on database developers (or power users comfortable with SQL) to build features, rather than requiring front-end developers or multiple specialized roles. For companies whose processes revolve around data, this can be empowering – the people who understand the data (DBAs, analysts) can directly implement application functionality without a translation layer. The flip side is that it asks developers to think in set-based, declarative terms (SQL) for tasks that they might normally do in imperative code; it's a different mindset, but one well-suited to business logic.

- **Faster Prototyping and Feedback:** The low-code aspects (auto-generated UI, immediate deployment) mean that an idea can be turned into a working prototype extremely quickly. Business stakeholders could get involved earlier, tweaking the schema or logic and instantly seeing changes. This tight feedback loop is reminiscent of modern low-code platforms, but unlike many of those, TSQL.APP still offers the **full power of SQL** for complex logic when needed [54]. It's a blend of productivity and flexibility that could reduce development time for enterprise apps.

- **Unified Lifecycle Management:** With everything in one place, maintaining the application becomes easier in some respects. Patches, upgrades, and audits involve the database primarily. Backup and recovery of the app is essentially the same as backup/recovery of the database. For businesses, this could mean less downtime and simpler disaster recovery plans – restore the database backup and you've restored the entire application (logic and all), not just the data.

- **Reducing Integration Pain:** In many business software projects, a lot of effort goes into making sure the front-end, back-end, and database schemas all stay in sync. Here, that issue is minimized because the database schema *is* the application, and the UI is derived from it. There's no ORM layer or API translation to worry about mismatching. This could lead to more robust systems with fewer data consistency bugs. As the platform itself puts it, **"positioning SQL Server as the application server"** blurs the lines but ultimately offers a more streamlined, data-consistent architecture [4].

It's worth noting that TSQL.APP is not just a theoretical idea; it's being used in real-world scenarios. According to its creators, the platform is already applied in building ERP, WMS, and other interactive enterprise systems, demonstrating **proven scalability in high-volume environments** [54]. This validates that the approach can handle the demands of serious business applications. It also means TSQL.APP is competing in the same space as both traditional enterprise software (like Oracle or SAP systems) and modern low-code platforms. In fact, it can be seen as an alternative to large-scale systems like SAP or to low-code tools like Outsystems/Airtable, because it offers the enterprise-grade depth of an SQL database with the ease-of-development of a low-code tool [55].

In summary, TSQL.APP's SQL-first, unified platform approach is quite different from the norm, and it has the potential to change how we think about developing data-heavy business software. By bringing the focus back to the database, it encourages designing applications around the core data and business logic from the very start. Developers and power users can work in one environment, using one language, to produce end-to-end functionality. This reduces the friction of multi-tier development and can lead to faster, more secure, and more maintainable enterprise applications [4]. For organizations heavily centered on data, adopting a platform like TSQL.APP could mean significantly accelerating development cycles while leveraging the robustness of proven SQL Server technology. It's a

bold re-imagining of application development, one that indeed *“transcends the framework”* to become a full-stack **application platform** grounded in the power of T-SQL [56] [57].

---

[1] [4] [6] [7] [8] [10] [11] [13] [14] [16] [34] [35] [36] [38] [45] [46] [47] [49] [50] [51] [52] [54] A Technical Overview of the SQL-First Application Development Platform.md
file://file_00000000a6806243b8645d47688fc434

[2] [5] [19] [32] [33] [53] [55] [56] [57] TSQL.APP is more than just a framework.md
file://file_00000000856061f482494a80de65c139

[3] [12] [15] [17] [18] [20] [21] [22] [23] [39] [40] [44] [48] what makes tsql.app special.md
file://file_000000005a78624696a89f388738aad7

[9] [24] [25] [26] [27] [28] [29] [30] [31] [37] [41] [42] [43] oct2025 - tsql_app_knowledge_from_schema_mcp.json
file://file_00000000f18c62468ea7d83563683d61