SOLUTION BRIEF

Pragmatic API Integration: from SDKs to Data Drivers

Pragmatic API Integration: from SDKs to Data Drivers

As data has become increasingly vital to core business functions, it has also become more decentralized than ever. Today, many organizations store critical data across a wide array of disparate systems. With the average company running 20 or more SaaS applications — and at least as many on-premises databases and internal apps — integrating, managing, and maintaining data connectivity presents a daunting challenge. Fortunately, several different solutions facilitate API integration, enabling organizations to ensure their data remains an asset rather than a burden.

As a developer facing integration challenges, you have many options to consider and many decisions to make. Assuming applications expose an API — either RESTful- or SOAP-based — developers typically choose between one of the following approaches:

- Direct integration: Call APIs/Services directly using low-level HTTP facilities and implement direct API connectivity.
- Work with language bindings (SDKs) provided by the application provider: These bindings abstract many web service details and expose an object-based representation of data.
- Integration middleware: This could be an ESB, an ETL product, or even an enterprise scheduler — with either application-specific adapters or generic web service support.

The best option depends largely on your current and future use cases, time and budgetary constraints, as well as on the larger context of current and future integration requirements.

If you have a contained tactical requirement — such as orchestrating a few operations and checking for success or failure, or retrieving a simple response for display — the lightest possible solution may be the best.

However, if you anticipate more extensive requirements, such as retrieving complex information from an application for further processing, you will likely consider a data-centric approach to integration. Using drivers for integration, like those from CData Software, represents a logical step-up for developers seeking to simplify architectural complexity and build connected applications more quickly.

The Data Driver: What Is It?

CData Software specializes in building standards-based drivers to ease integration with SaaS, NoSQL, and Big Data systems. Our principal offerings are delivered in the form of JDBC, ODBC, and ADO.NET drivers. In short, we make SaaS and other systems look like a database to any consuming application.

CData's approach to integration piggybacks on the ubiquitous support for and familiarity of ODBC, JDBC, and similar standards. By wrapping APIs in fully standards-compliant drivers, we open your applications for integration into any context where you can call that driver.

Users interact with our drivers as they would any other database driver. For users of analytics, visualization, reporting, integration and other products, this means "drop the driver in and go." For developers, pick a driver supported in your language of choice — JDBC for Java, ODBC for Python, ADO.Net for C#, or another .NET — and code against it as you would any other database driver.

Each of our drivers maps resources and operations exposed by an API to a corresponding relational model, with tables, views, and stored procedures.

Translating an API into an intelligible relational model requires human input because there are so many ways a REST or SOAP API might model the underlying application. So our development team designs these mappings around realistic use cases and feedback from our customers. This modeling effort and the resulting simplified programming model is actually one of the major value-adds associated with our SaaS drivers.

A full SQL-92-ready query engine is built into each of our drivers. The full power of SQL is available to query and modify application data and invoke actions in the application by making one or several calls to the API. This query engine has been shipping in multiple CData products for years and is stable, performant, and well-validated.

In general, the relational model exposes most, if not all of the data available in the API. SELECT statements retrieve data, and DML operations like INSERT, UPDATE, and DELETE modify data and, in many cases, execute operations within the application. Stored procedures are used judiciously to expose functionality not readily represented elsewhere in the model.

Service API modeling

General guidelines can help explain how the relational model and SQL statements can align with an underlying Web service API. For example, for a REST API, a developer might follow these principles:

- Tables and views correspond roughly to resource collections.
- Individual singleton resources from a collection generally correspond to rows in a table or view, with attributes mapped to columns.
- Nested values in attributes may be represented as JSON strings in a table column (the driver's SQL dialect supports JSON functions to manipulate these).
- Sub-collections may map to a table with a foreign-key relationship to the parent table.
- CRUD operations on resources correspond roughly to SQL statements, with HTTP verb GET aligning with SQL SELECT; POST, PUT, and PATCH with INSERT or UPDATE; DELETE with DELETE; and HEAD with a SQL EXISTS predicate.
- Stored procedures are used judiciously to expose operations and resources not otherwise easily represented.

Mapping a SOAP API may require more modeling effort, but the outcome is substantially similar.

SQL operations, especially complex SELECT statements, may not correspond one-to-one with web service calls. For example:

- An INSERT of multiple rows may translate into multiple web service calls to PUT each new resource
- A complex query that joins data will probably make one or more separate GET calls for each table resource collection

Note that, just as with the HTTP verbs, the SQL verbs you use may have effects other than simply storing data or "creating" a resource. For example, an INSERT SQL statement can post to a user's Facebook feed or send a Gmail message.

SQL Optimizations

More advanced applications — CRM and ERP applications, for example — may support their own query languages (like Salesforce SOQL) or at least offer API methods for complex operations like joining master and detail data. Wherever such features are available, the SQL query optimizer will generate calls that exploit them. If the data source does not support these capabilities, the Drivers internal SQL engine will handle the operations internally and transparently.

Advantages of SaaS Drivers

SaaS drivers offer several advantages over native language bindings and other lower-level means of consuming an API. Most developers already know how to interact with databases in their language of choice — open a connection; send SQL queries, DML and DDL; process result sets; and close the connection to a database full of small clients. Any native SDK that exposes language-level objects may require a new learning curve.

- You can easily browse and interact with the SaaS application data in your preferred IDE by simply installing the driver and connecting. Drivers work with Eclipse, IntelliJ, Visual Studio, and other major environments, as well as other products like Excel.
- SQL support becomes a big boost to your productivity once you get beyond the simple retrieval of resources. For example, consider an e-commerce SaaS application like Magento. To generate a list of your top-spending customers in each state requires joins, grouping, aggregate, and sort-and-filter. Working against native language bindings, if the API doesn't expose exactly that data, then you have to code parts of that logic yourself — or stage the records to a database for further processing.
- The CData Drivers support caching of application data in memory and in common databases or files, which minimizes the need to hit the API for every call. Repeated calls for the same data are transparent to the rest of your application (except when you choose to manage the cache manually).
- The Drivers support the SQL SELECT INTO command that can dump the results of any query to a CSV or other delimited file.

- The drivers translate your SQL into optimized calls to the underlying application API, taking advantage of application capabilities like joining and aggregating wherever the underlying API supports them. Without this, you would have to examine the API documentation to identify the set of API calls to use for best performance.
- You can customize the mapping of API resources and messages to the relational data model, gaining additional flexibility without having to code everything manually.
- If you move from language to language, which many of today's complex application development stacks require, the differences among SDKs for different languages may become a bigger obstacle. With CData Drivers, the data model and SQL exposed by the database is always the same, even across ODBC, JDBC, and ADO.NET. Plus, in your target language, you're always coding against the standard ODBC/JDBC/ADO.NET interfaces. You may not find the same consistency among SDKs for the same SaaS in different languages and environments. Likewise, you may find other organizations have embraced different data models if they coded the interfaces themselves.

Conclusion

Developers are often many times more productive when they build applications with drivers. At the core of most integration challenges, users want to query, aggregate, join, and summarize data.

The use of drivers for software integration can provide a richer, more productive overall developer experience versus working directly against APIs, frameworks, web service calls, or language bindings By leveraging common standards-based data interfaces like ADO.NET, ODBC, and OData, you can simplify integration processes and produce applications that are more capable, resilient, and easier to maintain, in a fraction of the time and expense.

Common Integration Approaches

Direct Integration

Pros	Cons
» Greatest control	» Longest integration time
» Simplified integration	» Lifetime maintenance
	» Little or no reuse across data sources

Good for very contained tactical requirements, such as orchestrating a few operations and checking for success or failure

Direct API and service integration is often the most challenging. The service implementations of APIs vary considerably, even with attempts to standardize. Disparate data sources leverage a wide variety of different web service protocols for authentication, security and data delivery.

Beyond the initial development, direct connectivity requires investments in performance tuning, testing and failover/redundancy mitigation. Additionally, SaaS APIs are evolving rapidly, and the lifetime maintenance costs of direct integration are substantial. Service APIs changes are often unpredictable and can force maintenance updates at inopportune times.

Although direct integration does offer the greatest level of control, the lifetime cost of integration can be significant.

SDKs

Pros	Cons
» Tightly bound data access.	» Lack of shared data model
» Simplified integration	» Limited vendor support
	» Little / no re-use across data sources

SDKs speed the integration process between client applications and APIs through embeddable platform/technology specific developer libraries. They typically provide tight coupling between SDK interfaces and the underlying APIs that they connect with.

While SDKs can speed development, many developers feel that SDKs lead to unnecessary architectural complexity. They are often managed and maintained by the community (open-source) or from an API provider, and they rarely share similar developer interfaces. This disparity provides little opportunity for knowledge reuse across SDKs and creates significant challenges mapping between APIs.

Furthermore, integration with SDKs often creates a host of challenges with dependency management, or "dependency hell." Too often, the SDK provider does not maintain parity between the SDK and the underlying API, forcing application updates to take advantage of new API features. These updates often change interface definitions, requiring recompilation and deployment.

Integration Middleware

Pros	Cons
» Abstract data connectivity.	» High cost
» Shared data model	» Greatest complexity
	» Vendor lock-in
	» Least developer control
	» Limited data source integration

Integration middleware has evolved as a core component of back-office integration. ESB and ETL products power connectivity between systems and often expose some data interfaces for extracting data or automating processes.

The drawback of these solutions is that they are often large, complex, and expensive. They shift the burden of integration management from the developer to corporate IT, typically requiring significant technology investment and financial resources to implement, secure, and maintain. This approach generally limits the ability of developers to build and deploy applications that depend on data services from these solutions.

Drivers: The Data-centric Approach to Integration

Pros	Cons
» Low learning curve (enabling fast time to market) » Consistent interface (common SQL-92 dialect) » Minimal documentation requirements (tables are self-describing) » Single common API » Insulation from underlying interface changes » Straightforward connectivity from all platforms/ developer tools (ODBC, JDBC, ADO, etc.)	» Not typically designed for event-based messaging operations.

Developers are often many times more productive when they build applications with drivers. At the core of most integration challenges, users want to query, aggregate, join, and summarize data. CData Drivers make this easy, offering a robust SQL-92 engine with support for deep query normalization and extremely complex queries, JOINs, GROUP BY, ORDER BY, and all filters (including ones with formulas).

The single consistent interface enables developers to quickly integrate with any supported data source. Because tables are self-describing, developers spend more time connecting data and less time hunting through documentation.

In addition, the CData Drivers are highly optimized for performance. Instead of pulling down large data sets and doing client-side SQL operations, they optimize their internal API calls and pass as much of the data processing as possible to the platform or data source. Only when the data source is unable to perform the required operation will processing be handled within the driver itself on the client. By offloading processing to the data source, CData Drivers will retrieve smaller data sets and benefit from substantially faster performance.

Additional Benefits

- Connectivity with common tooling (beyond development tools)
- A massive increase in API accessibility
- E.g. code-less connectivity from popular BI/ETL/Reporting tools

For a complete list of current Drivers and supported data sources, go to: www.cdata.com/drivers/

CData Software (www.cdata.com) is a leading provider of data access and connectivity solutions. We specialize in the development of Drivers and data access technologies for real-time access to on-line or on-premise applications, databases, and Web APIs. Our drivers are universally accessible, providing access to data through established data standards and application platforms such as ODBC, JDBC, ADO.NET, OData, SSIS, BizTalk, Excel, etc.

Our goal is to simplify the way you connect to data. We offer a straightforward approach to integration, with easy-to-use, data providers, drivers, and tools accessible from any technology capable of interacting with major database standards. This approach to integration allows businesses to realize the tremendous benefits and costs savings of integration while reducing complexity and expense.