System Evolution

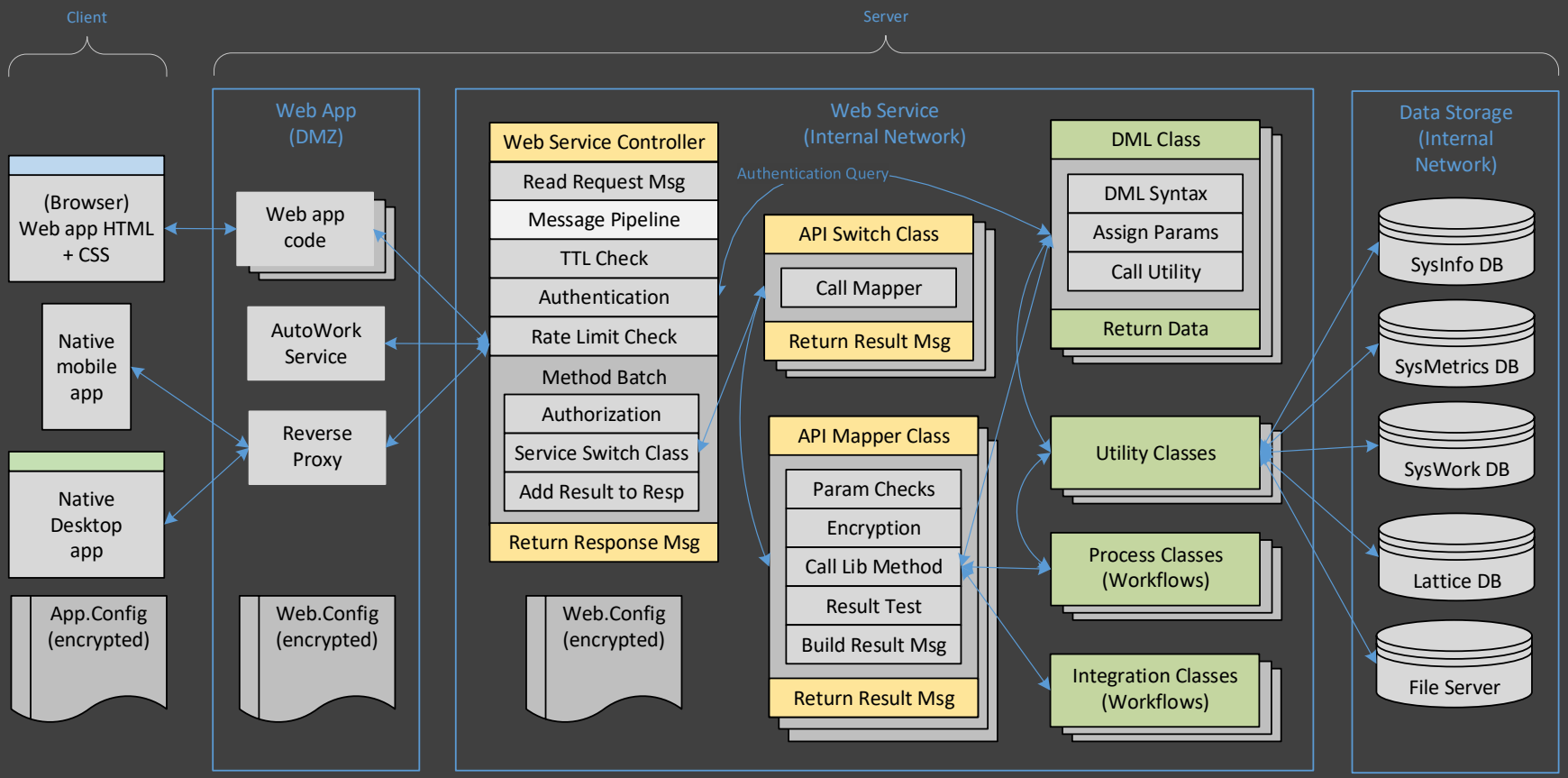
System evolution serves two main purposes: First, to enable software systems to be easily and inexpensively extended over time in order to meet the new and changing requirements. Second, these same capabilities can also be used to correct mistakes and eliminate technical debt from DGP software systems. In addition, evolution is often the fastest and easiest way to quickly build a new DGP system by appending new APIs, databases and UIs to an existing codebase.

Basically, system evolution allows software systems to be easily and inexpensively rewritten incrementally over time, without ever breaking backward compatibility, while the system itself remains in constant use. Few if any software systems currently have this capability, and its importance to businesses cannot be overstated.

DGP's simple hub-and-spoke modular architecture follows an immutable append-only convention, which enables new extensions and enhancements to be easily added (appended) to a system without ever breaking backward compatibility. This capability also enables fixes and elimination of technical debt to be added to the software system in the same way. The data-driven RBAC security system controls access to all the old and new functionality, allowing them to coexist within the same system. A gradual transition from old to new functionality can then take place for each client application and external integrated system, each at their own pace, while the system itself remains in constant use.

In the diagram below, the hub-and-spoke architecture can be seen as the external spokes of the client application endpoints on the left, the web service APIs in the middle as the hub, and the internal spokes of the database server endpoints on the right. A hub-and-spoke system is extended and enhanced over time by adding new UIs and databases to the system as new "spokes", along with new APIs added to the hub to support the new functionality. A two-tiered client/server system (thin client/fat server) is then formed by grouping the API hub and the database server spokes together into a server tier, protected by strong security.

The server tier itself is a modular multi-tier architecture that follows an immutable append-only convention which implements all changes to a system as new additions, rather than by making modifications to any existing code. This allows for changes and new functionality to be constantly added to a system without ever causing any breaking changes. This also enables zero maintenance work for API methods, test files and documentation for all API methods that have been proven to be working correctly.



The classes shown in yellow are the web service APIs, while the green classes are the shared internal libraries of code that do all of the actual work in a system. The web services are simply a secure way to allow remote access to the logic of the various code libraries.

The net result of following the immutable append-only convention for service evolution is that the logic of a system becomes a growing collection of APIs, API methods, and code libraries that gradually expands over time as the system evolves to meet the changing requirements of an organization. From a client application perspective, the DGP APIs are intended to imitate the namespaces of the .NET Framework itself. Remote client applications and integrated systems are then built to use the specific APIs

that they need selected from the growing collection of APIs in a given codebase, similar to how a .NET application only uses a small subset of the namespaces from the .NET Framework.

New systems can be built using the same evolution mechanisms to append the functionality of the new system as new APIs, databases and UIs added to an existing codebase. This will often be the fastest option to build new systems quickly and inexpensively. These same mechanisms also make it possible to "rewrite" a system incrementally, one piece at a time by adding new functionality that replaces existing functionality in order to remove technical debt while the system remains in constant use.

The databases that support all of the API methods must also follow the same immutable append-only conventions for the various elements of their schemas and also for core data. The functionality to manage the evolution of both the schemas and core data is built into the DBSetup utility, which enforces the various conventions via idempotent processes.

Experimentation

New additions appended to a system can be thought of as experimental prototypes that are added alongside the all of the existing functionality, never breaking backward compatibility. New prototype API methods are tested and monitored just like all other API methods in order to prove whether or not they meet all of the documented requirements, with good performance.

This overall process is based on the scientific method, which has been adapted to software engineering for DGP. Prototypes (at all scales) are the experiments. Tests and measurements of the API methods create the experimental data. DGP then uses "champion/challenger" head-to-head competitions between different alternative implementations to prove which one is able to deliver the best results.

The ultimate objective of the experimentation is to enable choices between alternatives to be made based on tests and measurements of competing prototypes in order to prove which one is best able to meet all the requirements, at the lowest cost, and in the shortest amount of time. Following this practice of experimentation helps to ensure that a system always evolves using the best available options during the useful life of the system (basically helping to prevent the creation of technical debt in the first place instead of fixing it after the fact).

Finally, the scientific method uses the concept of independently reproducible experiments and test results in order to catch any mistakes that may have been made in the experiments themselves. The testing and test harness applications built into DGP allow testers to verify the correct functionality and high performance of all API methods in each environment of a system. The frequent full regression tests run by many different people represents the independent reproducibility of the test results in practice.

In addition, the unit tests and performance monitoring that are permanently built into each API method allows all users to constantly verify the correct functionality and high performance of every API method in a system, as it is being used in production environments. For example, the results of these built-in unit tests and performance measurements are shown in the status bar of the DGPDrive application, and are also saved in the SysMetrics database for members of the RemoteMonitor role. This effectively delegates the constant testing and measurement of each production environment to that subset of users, performed automatically during their normal use of the system.