DGP Architecture and Design Overview

In order to help solve the problems documented in the various reports cited in the System Overview, it is important to understand that they are ultimately caused by weaknesses and flaws in current best practice architectures, frameworks, tools, and methodologies used to build all of those software systems that are currently causing such costly consequences for US businesses every year.

"We cannot solve our problems with the same thinking we used when we created them." – Albert Einstein

Based on the truth of this quote, a more effective software development methodology is needed that does not repeat the same mistakes of the current "best practice" solutions. As mentioned previously, DGP has been designed and built using a software development methodology, based on the Scientific Method, which evolved over many years of work in the R&D groups of several large companies. Each iteration of software development follows this simple process:

(Discover) → Define Requirements → Analysis and Design → Implement → Deploy → Test Deliverables vs. Requirements

DGP NFR Summary

DGP's documentation contains a set of Non-Functional Requirements (NFR's) that are a good baseline for the majority of business software systems. DGP's NFR's are described in more detail within their respective sections of the requirements documentation.

- Strong, effective security – refer to the security section of the documentation for more details
    - Strong network perimeter security
    - An identity store that provides data-driven Role Based Access Control (RBAC) authorization functionality
    - Simple, fast and reliable mechanisms for both account authentication and authorization to control access to system resources, functionality and data (based on the security data managed by the RBAC identity store)
    - Security controls for message TTL, account rate limits, failed authentication counts and lockouts, password resets, etc.
    - API-level encryption and decryption of data stored in select fields of database tables.
    - A scalable, transparent and easy to maintain mechanism to manage the encryption of system secret data (connection strings, system account credentials, etc.)

- High performance (.NET) – all API calls must return a result to the calling application in less than one second, end-to-end.
- High efficiency (.NET) – the web service APIs must maintain high levels of performance and low levels of memory usage over long periods of continuous use; in other words, no memory leak behavior, and no slowdown in performance over time.
- High availability – 100% uptime for the system with sufficient redundancy – individual servers or entire locations can go offline (planned or unplanned) while the system as a whole continues to function uninterrupted.
- High scalability – the system must be able to scale both processing power and data storage up to maximum levels that are well above the needs of most organizations, which allows systems to start small, grow as needed, and never outgrow the system’s original architecture.
- Processing node web service API methods should have an average server-side processing time of less than 10 MS.
- Processing node web servers should be able to handle a minimum of 100 requests per second per core.
- The ability to run all of the tiers of an entire system on a single computer is required for software development, testing, debugging, and to run small-scale production environments.
- Automated testing – testing built into the architecture of the web service APIs that allows both ad hoc and automated full regression tests (a necessary prerequisite for frequent/continuous deployments).
- Long duration load testing – the measurement of the end-to-end performance and resource utilization (memory, storage, network) over long periods of high-volume workloads.
- Automated processing subsystem – a scalable subsystem to run iterative processes continuously in the background, with full security, monitoring and recovery functionality; all automated processes are implemented as API methods; (optional subsystem).
- Simple deployments and rollbacks – the software must be easy to deploy and just as easy to roll back whenever necessary.
- Easy system evolution that always preserves backward compatibility – the constant extension, enhancement and fixes made to a system (evolution) must never cause breaking changes to the applications and integrated systems that depend on the API functionality.
- Continuous testing (and performance monitoring)
    - Unit tests permanently integrated into each API method, with the results returned in every API response message.
    - Server-side performance monitoring of each API method, with the duration returned in every API response message.
    - An API Tester test harness that runs full regression tests of every API method in a system, and can also be used to generate a synthetic workload for load tests.

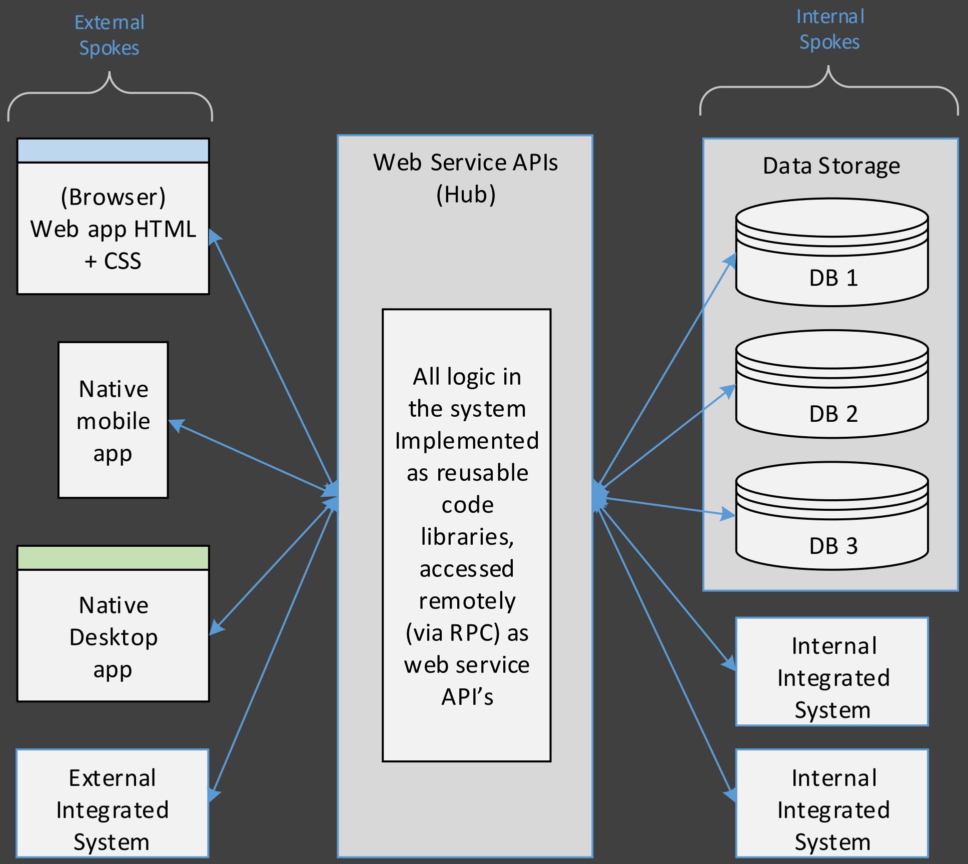
- Zero maintenance of API methods, associated test files and documentation pages once they have been put into production (thanks to the immutable append only conventions).
- System data management subsystem
    - Configuration data
    - Secret data (system account credentials, database connection strings, and other sensitive data) – this functionality is the equivalent of an HSM to protect sensitive configuration data in the system
- Data caching – caching built into the DGP web service APIs in addition to the caching in the database servers and web servers.

DGP Universal Architecture

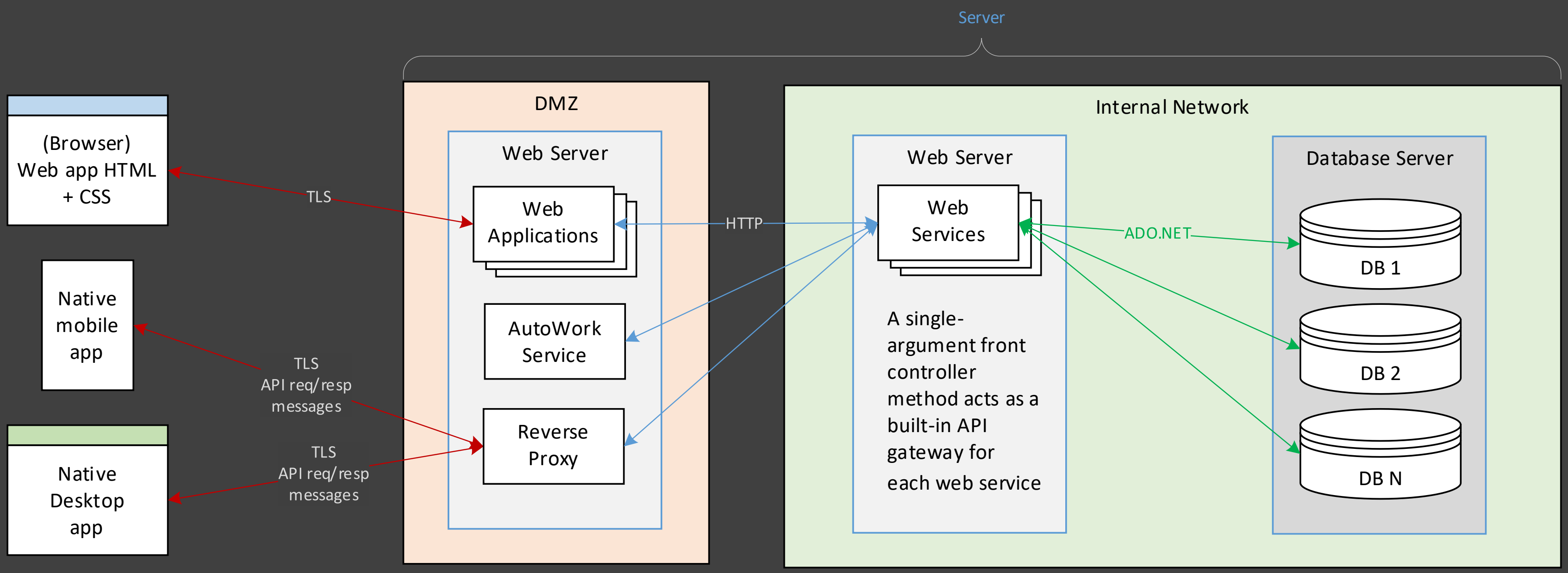
DGP's "universal" architecture gradually evolved in those R&D groups in order to make it possible to deliver functional prototypes as quickly, easily and inexpensively as possible. Fundamentally, it is based on Gall's Law, which states that "A complex system that works is invariably found to have evolved from a simple system that worked" (http://principles-wiki.net/principles:gall_s_law).

In practice, the implementation was made possible by using the hub-and-spoke architecture, which appends new UI spokes, data storage spokes, and new APIs to the hub of an existing base system. All of the UIs and functionality of the base system are reused by the appended prototype systems. Once the base system has been built, the main emphasis of this methodology becomes the definition of requirements, experimentation and testing (using APIs as prototypes) in order to prove which prototype delivers the best results, able to meet all of the requirements, at the lowest cost, and in the shortest amount of time.

As mentioned previously, DGP's universal API-centric architecture evolved over time in the R&D groups of two large companies, both of which placed a heavy emphasis on defining requirements and proving out the functionality of new systems using prototypes. The hub-and-spoke logical architecture shown in the diagram below has proven to be a good starting point for API-centric distributed systems. Hub-and-spoke is basically a simpler version of the port-and-adapter (Hexagon) architecture, delivering all the same benefits without the need for the added complexity of the ports and adapters themselves. This architecture is very compatible with modular multi-tiered distributed systems and allows for easy extensibility by adding more "spokes" linked to the central logic hub as needed. Adding more spokes will usually require the addition of more logic to the central hub, which is why a modular multi-tiered mechanism to add more web service APIs to the hub over time is needed.


External Spokes
Internal Spokes
(Browser) Web app HTML + CSS
Native mobile app
Native Desktop app
External Integrated System
Web Service APIs (Hub)
All logic in the system Implemented as reusable code libraries, accessed remotely (via RPC) as web service API's
Data Storage
DB 1
DB 2
DB 3
Internal Integrated System
Internal Integrated System

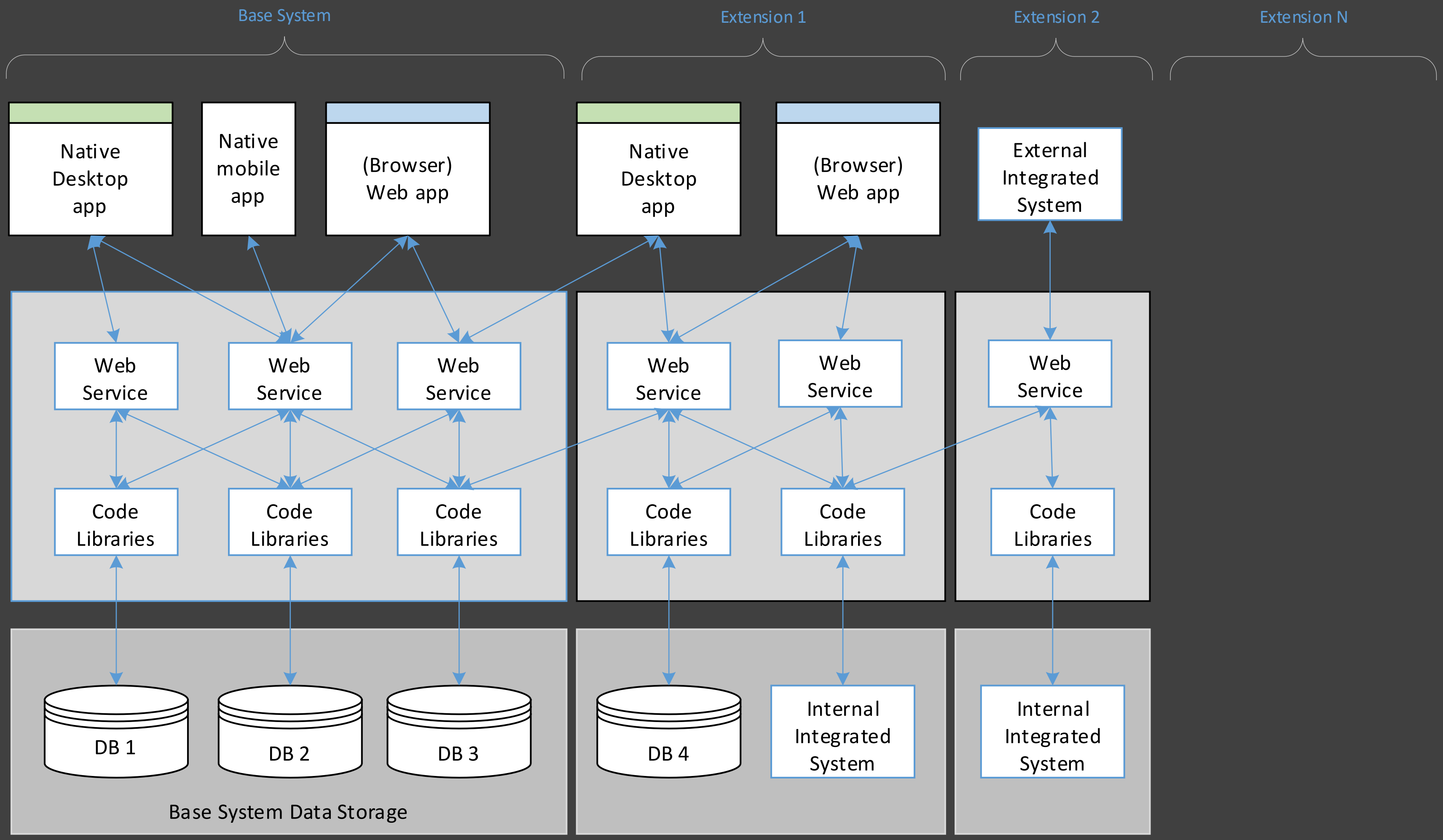
The hub-and-spoke logical architecture has proven itself to be a very effective foundation for building distributed systems, with a methodical and systematic mechanism to extend the system at a high level by adding new spokes to the hub, along with additional logic in the hub, as needed. The client application and external integrated systems are loosely coupled and only need to be able to create API request messages and read API response messages, which can be done by any one of the major programming languages. The messages are not treated as serialized objects, making them very generic and compatible without the need for any client SDK's.



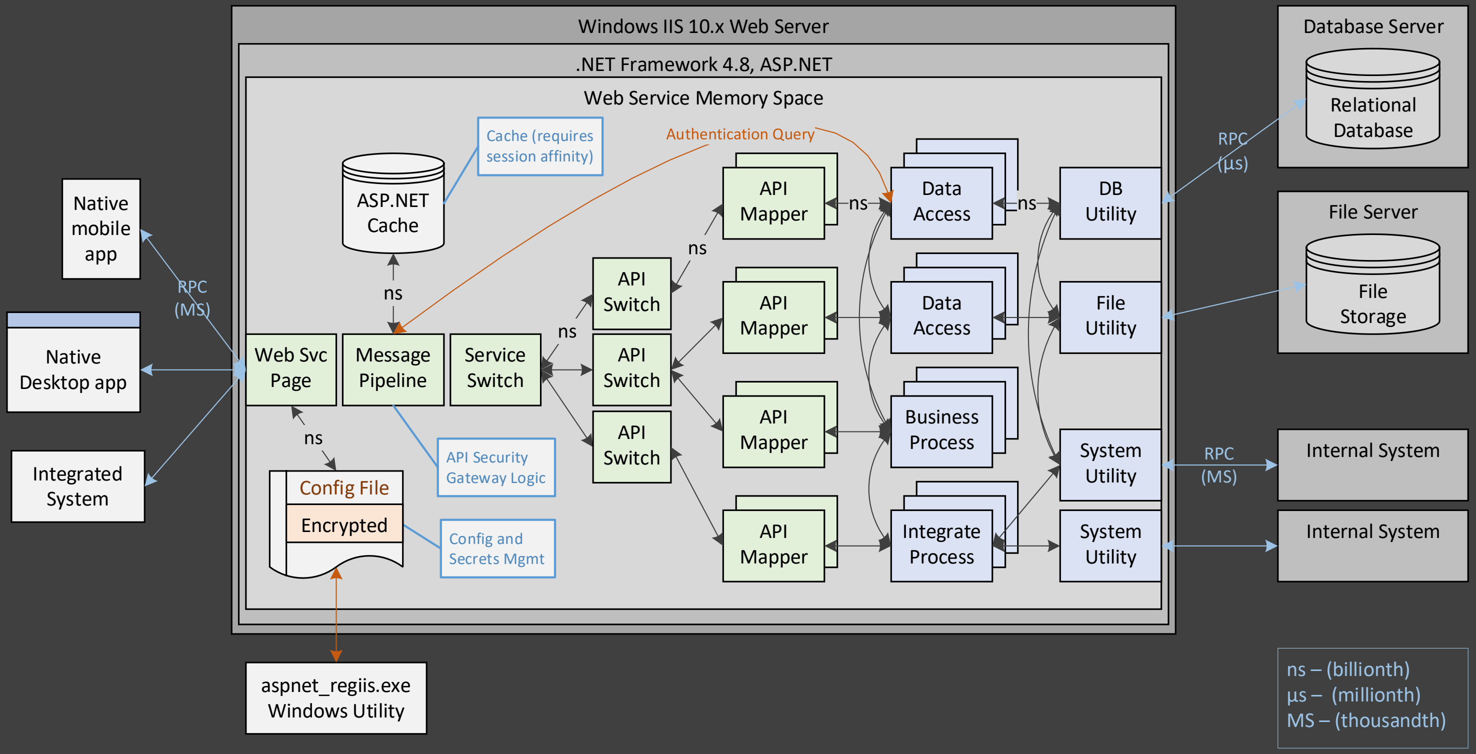
DGP's thin client/fat server architecture shown above groups the hub of system logic, implemented as web service APIs, together with the data in the internal server tier while the remote client applications are only used as a thin presentation layer. The natural grouping of the logic and the data in the server tier simplifies the processing of the data by the system logic. In addition, hosting both the logic and data on the servers inside the internal network significantly improves the overall security of the system.

The centralization of the system logic and data in the server tier changes the nature of the APIs themselves into RPC-style shared libraries of general-purpose functionality. This change enables other functionality to be integrated into the web service APIs. For example, DGP web services include the functionality of an API security gateway, data caching, configuration data, and secret data management subsystems directly into each web service, eliminating the need for all of those separate subsystems.

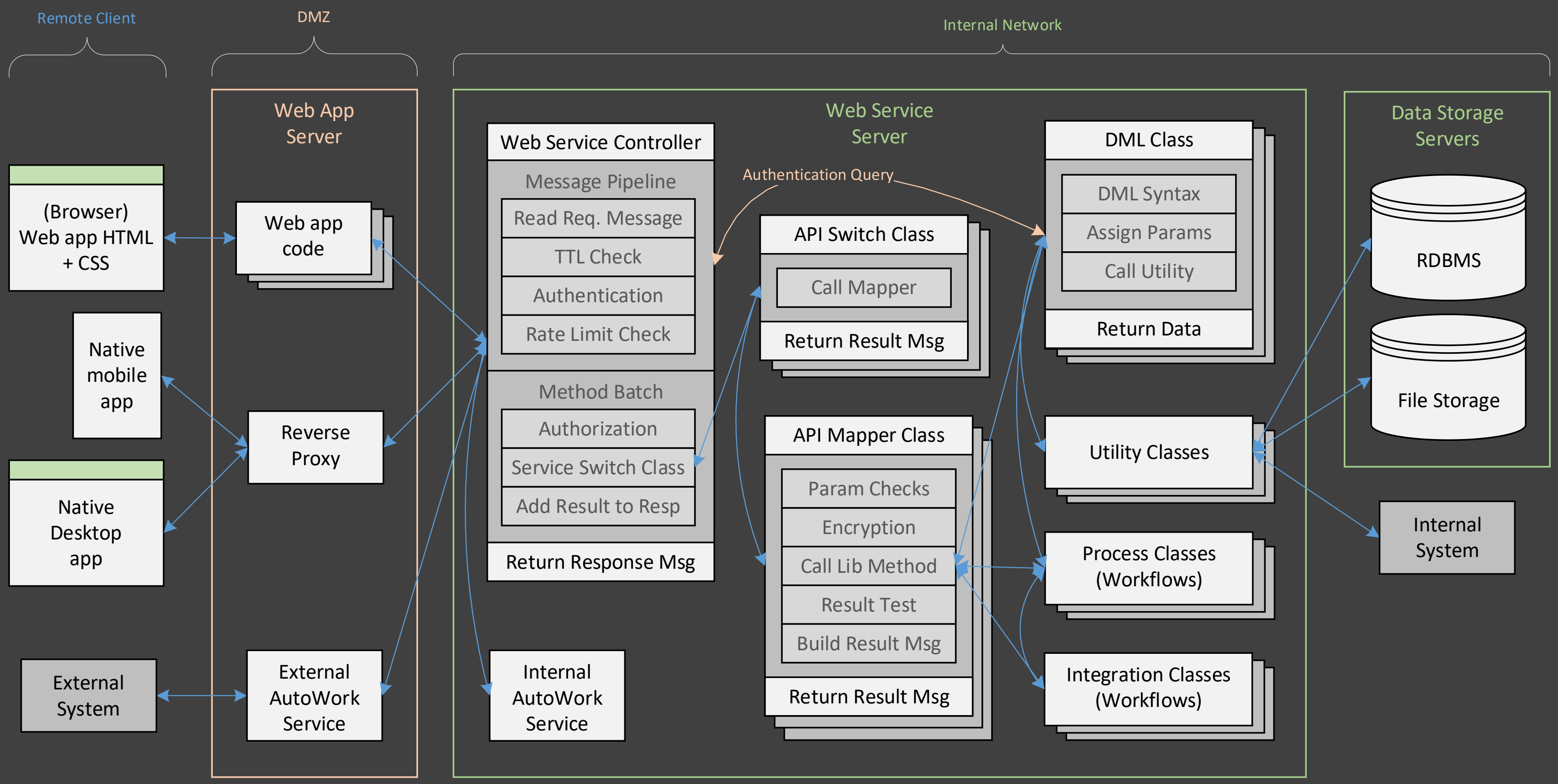
In this architecture, prototypes are built as new APIs, new API methods, and sometimes also with new data storage "spokes" as well. The combination of message-based APIs, a data-driven RBAC security system for authentication and authorization, plus the immutable append-only convention for APIs, database schemas, etc. makes each new API appended to a system automatically become a prototype by default as part of the normal system evolution functionality.



Modular Multi-tiered Hub-and-Spoke Architecture (Universal Architecture)



The diagram above shows the data flows within a DGP web service. The speed of the communication between remote systems (MS – thousandths of a sec), RPC calls made within a rack access switch (µs – millionths of a sec) and calls between libraries within a single memory space (ns – billionths of a sec) are also shown. Within the server, all of the modular logic of the system is actually implemented as reusable libraries of code (shown in blue) behind the web services. The DGP web services (shown in green) provide secure remote access to the functionality of the various code libraries via the message-based API methods. As much as possible, libraries call other libraries in the same memory space in order to significantly improve performance.



The diagram above shows additional detail within the various classes of the DGP web service. The simplicity of the architecture and code is indicated by 1) the fact that this low-level diagram showing all the classes and many of the methods of an entire DGP system fit on a single page, and 2) the implementation of all the server-side logic in the hub for a DGPDrive system totals approximately 12,000 lines of C# code when measured in Visual Studio. DGP places a lot of emphasis on producing very high performance, highly efficient native executables, which not only makes the centralized architecture viable, but are also instrumental in reducing the total cost + time needed to deliver high quality software systems.

The consolidation of the logic and data on the server also reduces the amount of data that is transported to the thin client applications, which only need to display pages of finished result set data to the user. This reduction helps to improve the both the performance and security of the system. Many of the OWASP API Top 10 security problems have to do with too much data being sent from the server to the client applications, with little or no security for the remote data within the fat client logic.

DGP Capabilities and Innovations

As mentioned in the System Overview documentation, the combination of a number of unusual design elements working in concert have made some very beneficial and innovative capabilities possible. In the DGP architecture, almost all features and functionality depend on the use of the message-based RPC APIs. The creation of new APIs is greatly simplified by the use of the immutable append-only convention, which renders the API methods, test files and documentation pages immutable once they are used in production. The message-based APIs, integrated data-driven RBAC security system, and the immutable append-only convention in particular make the zero maintenance, continuous testing, system evolution and competitive experimentation capabilities possible. The use of several “micro-batching” mechanisms are also extremely important in order for all logic in a system to be able to be implemented as web service API methods. Server-side pagination integrated with flexible search criteria allow queries of data with large result sets to be broken up into pages of data returned to the client applications one page at a time. Processing large amounts of data (using DGP data replication as one example) is similarly broken up into micro batches of work processed sequentially (managed by the AutoWork automation subsystem), each iteration of which can be implemented as an API method call.

- Security

DGP’s improved security depends on the message-based RPC APIs, which use a single argument front controller pattern that reduces the exposed attack surface of the web service down to a single method which accepts a single input parameter. Account credentials in the form of a security token is contained within each API request message, and this is used to provide both account authentication and method authorization for each of the API methods called by a request message. To accomplish this, the controller method feeds all API request messages into a message-processing pipeline that encapsulates all of the security features of an API security gateway. The message processing pipeline authenticates each API request message and authorizes each internal API method call using the data-driven RBAC that is part of the custom identity store. Each API request must pass the following checks before it is allowed to call any of the internal library methods:

- Message TTL check (prevents replay attacks)
- Account authentication
- Failed authentication count check (prevents brute-force dictionary attacks)
- Account authorization for each individual internal method call
- Account API call rate limit check (helps to mitigate DOS attacks)
- (optional) Integrated message encryption

The security system also includes API-level encryption of sensitive data fields in the database, and an integrated secret data management system that provides most of the same functionality as an HSM. Details about the design, implementation and proof of the security system functionality can be found in the Security documentation. The ability of DGP's RBAC security system to provide fine-grained control over which methods can be called by which accounts is crucial to the system evolution functionality.

- System Evolution

The objective for the practice of system and service evolution is to design software systems so that they enable constant extensions, improvements, and fixes to be added to a system without ever causing any breaking changes to the applications and integrated systems that depend on the API functionality. DGP system evolution enables software to be incrementally rewritten to add new functionality, fix mistakes and remove technical debt, while the system remains in constant use, and without ever breaking backward compatibility. The immutable append-only pattern used for APIs, code libraries and database schemas is one of the keys to achieving these objectives in practice. In turn, the system evolution functionality is a crucial part of both the zero maintenance and prototype experimentation capabilities.

At a high level, system evolution allows DGP's centralized APIs to behave much like a server-based version of the .NET Framework, whose collection of APIs is shared by all the various "spokes" of the hub-and-spoke architecture. Each client app or integrated system only uses the specific API methods that they need from the ever-growing collection of APIs in a DGP system. Details about the design, implementation and proof of the evolution functionality can be found in the Evolution documentation.

- Zero Code Maintenance

Another advantage of following the immutable append-only convention is the fact that no maintenance of existing DGP code is required. All DGP functionality is built as API methods, and once an API method is used in production it becomes immutable, along with its corresponding API Tester test file and documentation web page. The only exception to this rule is bug fixes, which should never affect the signature of the API method, and very rarely make it into production due to all of the testing performed in the lower environments. The ongoing extensions and fixes of system evolution are implemented as new API methods appended to the code base instead of merging changes into any existing methods. Authorized access to these new methods is controlled by the RBAC security system in each environment. Obsolete methods can be deprecated once it is proven that no applications are using them. This mechanism is not only for system evolution, but is also used to eliminate technical debt, as new improved methods can be built to replace older methods that contain technical debt limitations.

A very important side effect of zero maintenance is that small teams of developers and testers are all that is ever needed for a software system, no matter how large it may eventually become over time. The team only needs enough people to keep up with the workload of adding new functionality to the system and testing it using the API Tester test harness application. All existing API methods are constantly tested to verify their correct functionality, and are never modified (or broken) by new development. The same is true for their associated test files and documentation pages. The combination of the zero maintenance and continuous testing innovations are responsible for the largest reductions in software development costs for DGP systems.

- Experimentation (functional prototypes)

Prototypes are built by adding new APIs or API methods to a system using the system evolution and zero maintenance capabilities. The APIs themselves consist of two parts. First, the web services provide a mechanism for secure remote access to the functionality of a system via API request and response messages. Second, the reusable libraries of code running behind the web service APIs are used to implement all of the actual functionality in a system. This allows that functionality to be called remotely as an API method when needed, or be called as a library method within the same memory space whenever possible.

In addition, the single-argument front controller pattern provides one of the few layers of abstraction in the system, allowing the naming convention of the APIs to be completely independent of the naming convention used for the libraries of code behind them.

- Continuous Testing

Continuous testing is another of the major innovations found in DGP. Every deliverable must be tested and measured to verify that it is able to meet all of the documented requirements of a system. This change in emphasis from focusing on the source code and development process to instead focusing on testing and measuring the deliverables (results) helps to significantly improve the quality of software systems in and of itself.

The message-based APIs provide two-way communication between the client application and the server, which allows more information than the normal method call and return value to be sent back and forth between them. The mapper methods of the APIs contain the equivalent of unit tests permanently integrated into each method. When the results of the internal method call are returned, the mapper method evaluates the return value to determine if it executed correctly or not (similar to a unit test assertion). The results of these tests built into each mapper method are returned to the client application as part of every response message.

In this way, every user is effectively testing the API methods of a system continuously, every time an API method is called. The server-side performance of each API method call is also measured, and those results are similarly returned in each API response message. Plus, users that are members of the RemoteMonitor role automatically save these remote end-to-end test results and performance measurements to whichever location they are connected to at the time. This functionality is able to collect very useful real-time metrics from production systems without adversely affecting those systems due to the testing and monitoring workload.

In addition, the API Tester test harness application is able to remotely run full regression tests of all of the API methods in a location. The test files used by the API Tester application contain a series of both positive and negative tests for each individual API method, and can be run by any account that is a member of a test role in a given environment (but only for authorized methods). The results of the end-to-end full regression tests can optionally be saved to the SysMetrics database for further analysis as desired. Finally, the API Tester application is also able to run load tests of a DGP system. Allowing many people to easily run the tests is part of the effort to ensure that the tests and measurements are independently verifiable by both the users and testers of each DGP system. Details about the design, implementation and proof of the testing functionality can be found in the Testing documentation.

- 100% System Uptime and Automatic Recovery

DGP’s innovative functionality depends on its centralized modular multi-tier hub-and-spoke architecture. This type of distributed system has been used successfully for decades (in the form of a load-balanced web server farm with a database cluster back end), and has proven itself over time to be one of the best logical and physical topologies for software systems currently available. The following URL provides a good overview of distributed systems, grid systems, their similarities and differences: https://www.toolbox.com/tech/cloud/articles/distributed-vs-grid/

This topology provides incremental horizontal scalability and n+1 fault tolerance for the processing nodes in its web server farms, but its data storage cluster redundancy will generally only work within a single location. However, basing DGP on such a well understood foundation is a key factor helping it to remain simple, easy and inexpensive to build and maintain. Also, this type of simple distributed system is still the best solution when 100% system uptime and all of the redundant hardware and software are not needed.

Standard storage systems (RDBMS, NoSQL, etc.) are able to replicate data between the nodes of a cluster within a single location, while designating a single node in the cluster to be writable in order to provide strong data consistency. Unfortunately, they are usually not able to replicate data reliably between separate locations. However, the ideal functionality desired by businesses for many software systems would allow multiple separate locations to be active and writeable at the same time, which provides true 100% system uptime, and also effectively eliminates the need for Disaster Recovery and Business Continuity processes.

DGP has an (optional) omnidirectional data replication subsystem that has been specifically designed to continuously replicate data between multiple active, writable locations. As a result, DGP can be thought of as a standard distributed system if standard data storage is used, and a distributed grid system if the optional DGP automated processing and data replication are used. There is also a way for SQL Server Availability Groups to deliver the same 100% uptime for large-scale systems, which can be used in place of DGP's replication functionality as desired.

While DGP's asynchronous data replication represents an AP system with eventual data consistency in terms of CAP theorem, strong data consistency can be achieved by only using a single location at a time (having a primary location and backup location), which means that the issue of eventual data consistency only affects data replication between the primary and backup locations. The backup location remains active, writable, and ready to take over for the primary at a moment's notice – eliminating the need for DR or business continuity processes. Details about the design and implementation of the data replication, verification and repair functionality can be found in the Availability documentation.

- Performance and Efficiency

High performance APIs with good efficiency are an essential prerequisite for the use of a centralized API-centric architecture. The server-side processing duration for DGP web services averages roughly 10 MS, assuming no caching is done at the API level and 80% reads vs. writes. This duration decreases to an average of 5 MS or less as more and more levels of cache hits take effect. At that point, the end-to-end duration of API calls measured by the client application is basically the same as whatever network latency exists between the client and the server. High efficiency ensures that high performance does not degrade over time. Details about the design, implementation and proof of the high-performance APIs can be found in the Performance documentation.

- Scalability

The basic load-balanced stateless web server farm provides excellent incremental horizontal scalability. For data storage scalability, along with the normal vertical scalability achieved by adding more resources to the host server are database shards, which allow database storage to scale out horizontally for some (but not all) database schemas. For the other schemas, the ability to scale up database servers to have 10's of TB of solid-state storage, with over a million IOPS and data throughput measured in GB per second is well beyond the maximum requirements of most organizations. Details about the design and implementation of the database shards can be found in the Scalability documentation.

- Monitoring

The two-way communication provided by the message-based APIs allow for server-side API performance measurements to be returned in every API response message. For example, in the DGPDrive system, the total end-to-end performance of each API call is measured by the application, the server performance is returned in the API response message, and both are displayed to the user in the status bar as they use the DGPDrive application. For users that are members of the RemoteMonitoring role, these performance measurements are saved to a database in the location to which they are connected. The number of members of the role control the volume of performance metrics data being collected. In this way, the functionality and performance of production systems are constantly being tested measured during their normal use. The saved metrics data, along with saved error data from the consolidated logging, provides a good level of system observability. Details about the design, implementation and proof of the monitoring functionality can be found in the Monitoring documentation.

- Logging

DGP includes global logging of information, errors, and exceptions across all of the tiers of the system. The logging itself is a two-stage process that writes entries to both the local event viewer, and also calls an API method to log the error to a database in the location to which it is connected. Under most conditions, the error data will be successfully stored in their respective databases, but when data access is part of the problem, a way to monitor the local event viewers for the error data on the servers (and to a lesser extent the remote clients) is needed. Verbose information about system events, processes, etc. are generally logged to separate log files. Details about the design, implementation and proof of the logging functionality can be found in the Logging documentation.

- API-based Automated Processing

DGP has a subsystem to perform continuous iterations of automated work, such as data replication, data verification, data repair, etc. All of the automated processes are implemented as API methods so that they can be run as DGP accounts with full API security. Special database tables are used as work queues that control the execution of each iteration of complex processes, plus the functionality of the automated processing schedule itself are also implemented as API methods. Multiple windows services acting as schedulers are then able to distribute the automated processing workload by calling the API methods to claim process iterations, and then calling the one-way API methods that do the work of each process iteration. Details about the design, implementation and proof of the automation functionality can be found in the Automation documentation.

- Deployment and Rollback (CI-CD-CT)

The simplification of deployments also depends on the immutable append-only pattern for new application versions and database schema changes. The DGPDrive application is written in C#/.NET, and uses folder-based deployments for the client applications, web application, and web service. In the .NET Framework, deployments are based on folders. The basic concept for the immutable append-only convention at this level is that release subfolders are never merged, but coexist side-by-side with previous release subfolders. if problems are encountered with a new subfolder, rollbacks simply point the application executable path back to a previous subfolder. Older subfolders can be removed after they are no longer needed for rollbacks.

Changes to database schemas must also follow an immutable append-only pattern, which is handled by the DBSetup utility that applies the changes to each database in each environment as needed. Rollbacks of the additions appended to database schemas are not supported. Details about the design, implementation and proof of the deployment functionality can be found in the Deployment documentation.

- Total Cost and Delivery Time Reductions

The total cost of a system is reduced by simplification, high performance, and by using less resources of all types (and in some cases substituting less expensive resources for more expensive ones). Delivery timelines are reduced by doing less work (simplification) and by automating simple, repetitive tasks whenever possible. One of the biggest reductions in work is due to the zero code maintenance aspect of the immutable append-only conventions. Zero maintenance not only eliminates a large amount of work, it also means that small software development teams are all that is ever needed (even for large systems), due to the fact that the teams only have to be large enough to build and test ***new*** functionality that is added to a system during each iteration of development. None of the existing code or methods or test files or documentation pages ever need to be touched once they have become immutable in production. Details about the reduction in both TCO and TTM and their proofs can be found in the TCO/TTM documentation.

Summary

In summary, DGP represents an effort to 1) provide SMBs with the ability to inexpensively build their own custom software systems that have all of benefits of a distributed grid computer, and 2) helping to solve most of the problems related to poor security and low software quality described in the supporting documentation from CISQ, OWASP, etc. It does all of this at a low cost and with short delivery times due to the simplicity of its architecture, code, the widespread use of the immutable append-only convention, as well as the use of rapid prototyping and rapid application development techniques.

In addition, DGPDrive is released as a free open-source project that includes the documentation web app. This web app contains all of the requirements, architecture, design, implementation, setup and configuration documentation that explain the details of all DGP and DGPDrive functionality. This documentation is updated to stay current with every release of the open-source code.

Finally, the DGPDrive system 1) proves how well all of DGPDrive's innovative functionality actually works in practice, 2) demonstrates the high performance of the web service APIs and the system as a whole, and 3) provides working examples of open-source code that show how custom software systems can be built by appending new functionality to the DGP code base.