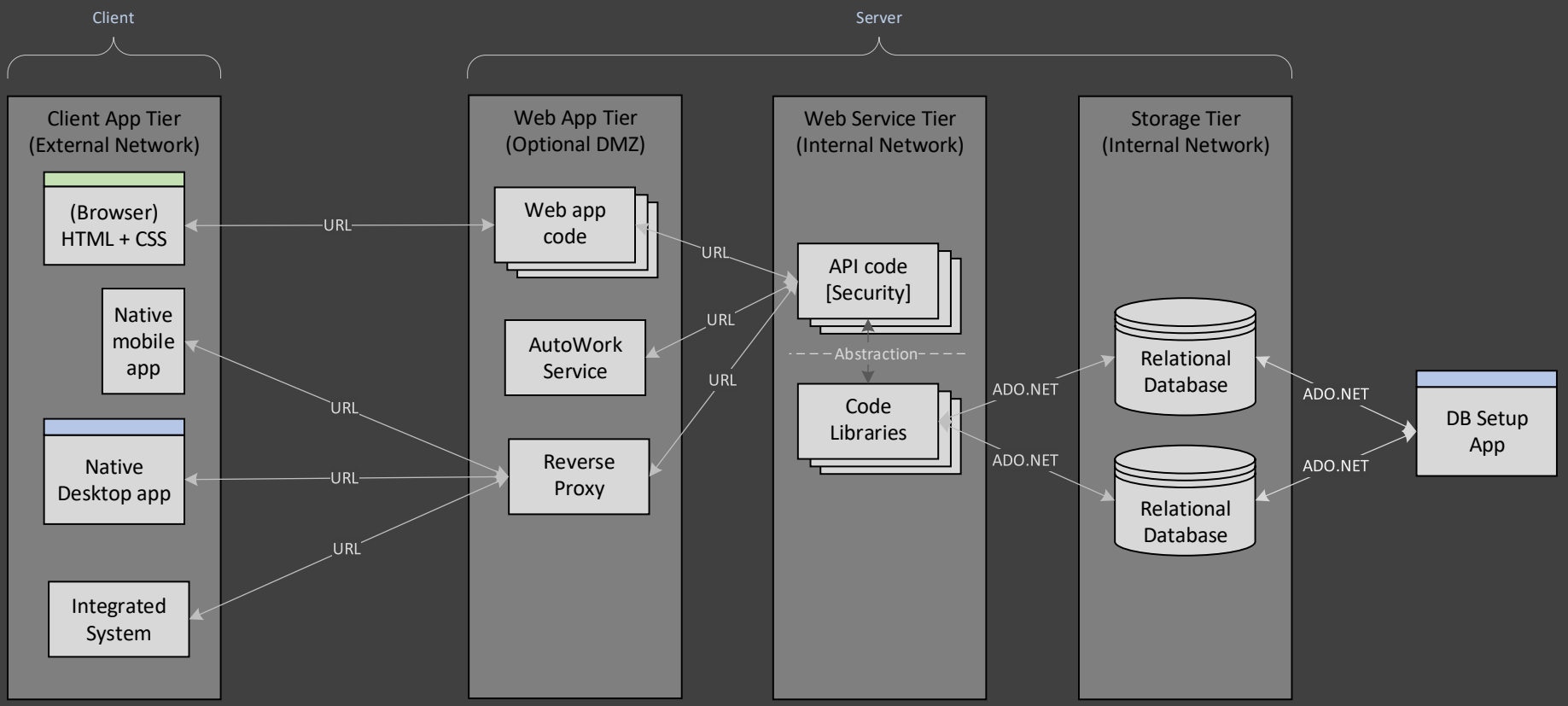
Modular Multi-tier Architecture



DGP uses a modular multi-tiered architecture that represents an optimal middle ground between the simplicity of a monolithic architecture and the scalability/complexity of a micro-services architecture. The multi-tier architecture is able to scale up and out to levels beyond the needs of almost all businesses, while also being able to collocate all of the tiers onto a single computer at a small

scale. Systems will generally start out as a single computer in each distributed location and only add more processing nodes, storage nodes and infrastructure to each location if and when it is needed to accommodate the growth of the system.

The structure of a DGP system is similar to the IIS web server. Each IIS web server is a container for one or more web sites. Each web site is a container for one or more web applications. Each web application is a container for one or more virtual directories. Following that same basic pattern, each DGP system contains one or more environments. Each DGP environment contains one or more locations. Each DGP location contains one or more computers (or VM's).

From a high level, each location is a two-tier client/server architecture that consolidates as much of the logic as possible into the server tier, which act as the hub in a hub-and-spoke topology. Logically the server has two tiers, one for processing work (web services) and the other for data storage (database servers). The DMZ is an optional physical tier added primarily for network security, so the real hub of the entire system are the web service API's and the reusable libraries of code sitting behind them.

Each tier is logically isolated from its adjacent tiers, and only communicate with the others using various types of RPC. This isolation allows each tier to evolve and scale independently of its adjacent tiers, adding new features and functionality on its own timeline. Another important aspect of this architecture is that all of the RPC's are IO bound. This allows for the very efficient use of thread pool threads, managed by the Task Parallel Library, combined with IO completion ports – all of which are managed automatically for a DGP system by the .NET Framework CLR itself.

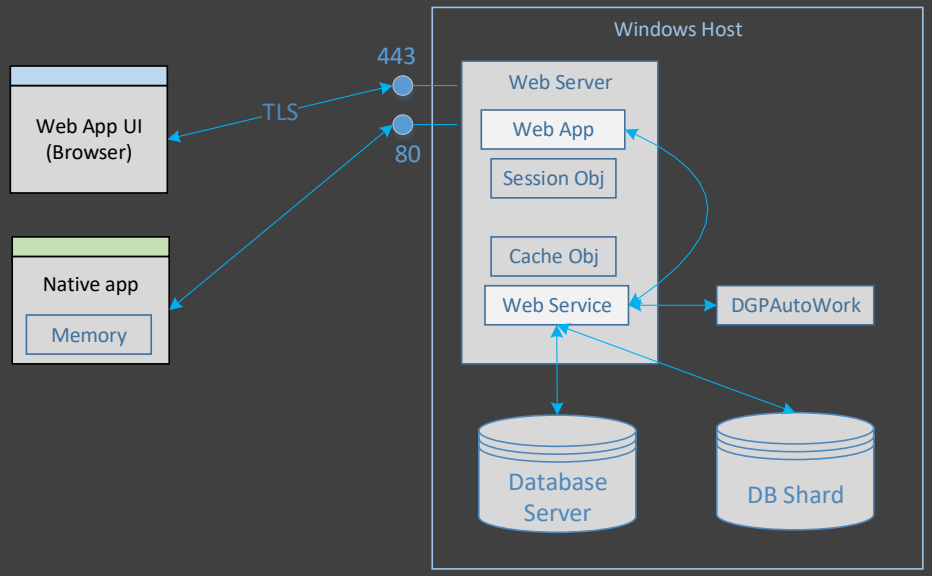
Virtually all of the functionality, along with all of the data in DGP systems are consolidated in the server tier. This provides several advantages:

1. Client applications can be built much more quickly and at a lower cost, since the client apps themselves contain little or no logic and are just a thin presentation layer. This also makes testing the client apps faster and easier as well.
2. Keeping all of the logic and data in the server tier greatly improves the overall security of the system. Sensitive data never needs to leave the centralized, private and well protected server tier.
3. Since all of the work in a system is done in the server tier where both the logic and data are collocated, client applications only need to collect input data from users and display finished results of the server-side work to the users.
4. It is much easier to debug and fix centralized functionality compared to logic embedded into remote applications.
5. It is much easier to maintain and deploy centralized web services compared to installing/updating remote applications.

The primary problem for this type of centralized functionality will usually be the slow responsiveness of applications that must call remote servers for every UI action. In fact, this type of centralized architecture is only feasible when the web service API's that are being called are extremely fast. Fortunately, DGP web services focus on high performance and efficiency, and those capabilities enable the use of this type of centralized architecture.

The minimum standard for UI performance is that *a UI must display the results of every action and also be fully interactive in one second or less.* A practical example would be when a user clicks a button in the UI, the app responds by displaying a new screen of data, and after displaying the data the app is then ready to immediately respond to the next user action – all in one second or less.

The reasons why one second is the important threshold are explained very well in this video from a Google engineer responsible for mobile and web app performance: https://www.youtube.com/watch?v=Il4swGfTOSM. These performance standards are applicable to all types of applications, whether they are web, native, or mobile.

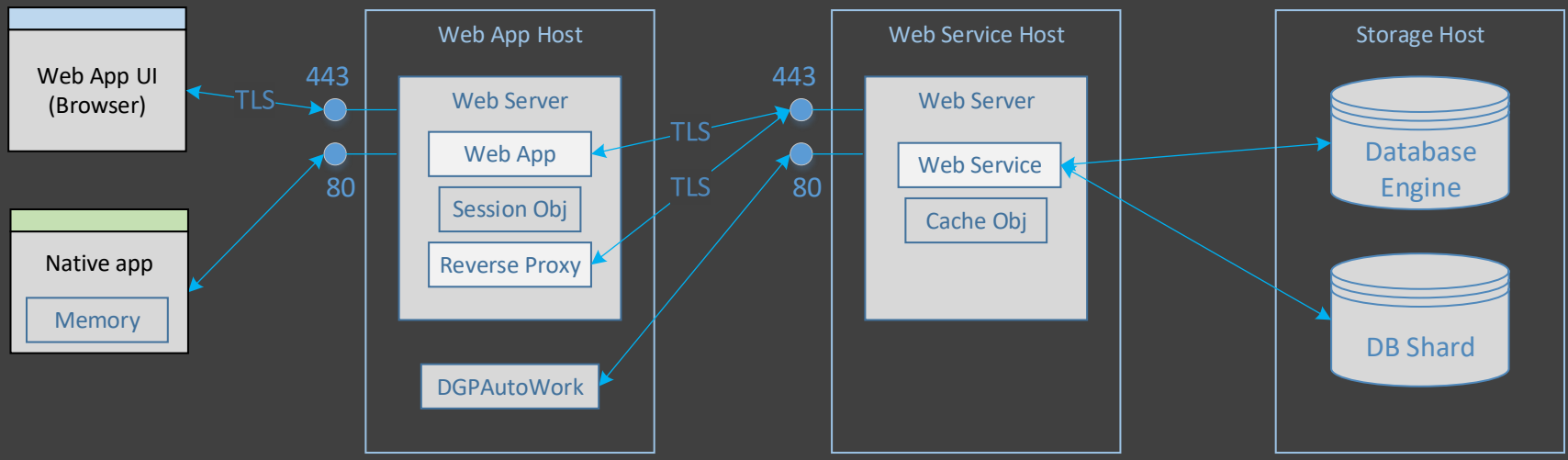


TLS

TLS is used to encrypt the API request and response messages during transport, but it is not needed for all environments. The use of TLS or HTTP to communicate between the tiers is handled by the web server and is transparent to the web apps and web services, with no differences in their functionality. Establishing TLS connections are an expensive operation, so reusing those connections once they have been established is important for good performance. Internally, either TLS or HTTP connections can be used, depending on what level of security is needed for a specific system. An encrypted web service controller is also available for native applications that can be built to contain the message encryption and decryption logic, and are used with standard HTTP connections.

Caching

Both native apps and web apps cache information about the user and the web service they are connected to. The native app caches that data in its own local memory, while web apps cache the same data in their Session object. Static data is stored in the App.config or Web.config files, which are read into memory when the app is initialized.



Secret Data

Static settings data for each tier is stored in the app.config and web.config files, which are read into memory when the app domain is initialized. Some of the settings will be secret or sensitive and require some type of protection. The AppSettings section of app.config and web.config files can be encrypted/decrypted using the aspnet_regiis.exe utility. This encryption functionality is used to protect secret data such as account credentials, connection strings, encryption keys, etc., instead of using HSM's or other similar additional subsystems. This type of distributed secret management solution works equally well for a single server, small scale systems, and larger systems, and is generally going to be much more scalable and fault-tolerant than an HSM or other type of cluster – however, centralized, global management tools for system secrets can be used in DGP systems as needed.

Reverse Proxy

When firewall rules are used to create a DMZ separated from the internal network, some type of reverse proxy is needed to allow native apps to call the web services hosted in the internal network through the DMZ. DGP uses reverse proxy pages that function in a similar way to the ASP.NET web app pages. They accept the API request message from the client app, create a new connection to a web service on the internal network, and forward the API request to the web service. The web service response is then returned to the app that called the reverse proxy page. Using these reverse proxy pages with load balanced server farms requires session affinity for both the DMZ and Internal network load balancers.

Load Balancer

Due to the use of session objects by the web applications, and cache objects by the web services, it is important for load balancers to provide session affinity (sticky sessions) for existing sessions on the web servers. Session affinity is also needed to reuse existing TLS connections as well. By default, the reverse proxy pages chain together two HTTP connections, first from a client app to a reverse proxy page in the DMZ, and then from the reverse proxy page to a web service in the Internal network. One or both of these connections can be encrypted using TLS. If DGP message encryption is used, they should both be standard HTTP connections.

Web App Functionality

Current DGP Lattice systems only use the web app tier for the documentation portal and simple reverse proxy pages. Other systems that did not use hybrid file storage and file transfer functionality could be written as web applications instead of native .NET client apps. In that case, Server Side Rendering (SSR) web apps can perform very well using the existing web service API's, with good security. Going forward, some of these types of web apps will be added to a DGP reference system, although Flutter apps may eventually be used for both native and web applications.

DGP web apps use Server Side Rendering (SSR), and cache the same data as native apps in the session object of the web app server. Web services can optionally also cache information in the cache object of the web service server. This caching functionality built into Windows and the .NET Framework eliminates the need for a caching subsystem such as Redis to be added to the system architecture. This type of distributed caching solution works equally well for a single server, small scale systems, and larger systems, and is generally going to be much more scalable and fault-tolerant than a Redis or other type of cluster – however, centralized, global caching tools can be used in DGP systems as needed.

Web Service Functionality

- Read API Request Messages
  - Accept HTTP/HTTPS POST of API request messages to the web service API endpoint
  - Read properly formatted API Request messages (XML fragment) as a memory stream
  - Check TTL of each API Request message

- System Security
  - Single-argument front controller web services expose the minimum possible attack surface
  - MIME type limited to only text/xml (no multipart form file uploads allowed)
  - Readers limit API Request messages to XML fragments only (no XML headers allowed)
  - Readers limit the size of request and response messages to 64K
  - Readers check the TTL of each Request message

  - Verify HMAC Hash Values used to Authenticate the user account in each Request message
  - Check the failed authentication count (5 consecutive failures results in account lockout – configurable)
  - Check the account rate limit count of method calls per minute (configurable per account)
  - Authorize each individual method call within the Request message batch
  - Create HMAC Hash Values used to Authenticate the web service to the client app in the Response (Login method only)
  - Create properly formatted API Response messages (XML fragment) returned to the client app

- Other Web Service Functionality
  - Server-side pagination of SQL query results
  - Enforcement of DataGroup security within select data access methods (similar to RBAC authorization of API methods for shared data – FileStore uses this mechanism to control which accounts can see which folders, files, etc.)
  - Cached UserInfo object to improve web service performance
  - Cached user account rate limits

- Error Handling and Logging
  - Logging to the local Event Viewer
    - All errors and exceptions are first written to the Event Viewer on the computer where they occurred
  - Logging to central database tables
    - All errors and exceptions are also written to the DGPErrors table in the SysMetrics database
    - When external, they are written by calling a web service API method (RemoteErrLog)
    - When internal, they are written by calling a data access method (ServerErrLog)
    - Errors and exceptions that may occur while writing to the SysMetrics database are themselves written to the local Event Viewer

The API request messages are limited to a maximum of 64K in size by the.NET XML reader in order to insure that the request messages will remain below the .NET 85,000 byte limit and avoid being stored in the LOH.

API Request Message

```
<ReqMsg>
  <UserName />                  - DGP system account name
  <ReqID />                     - unique ID created by the client app for each request message, and echoed back in the response
  <ReqToken />                  - HMAC hash of the Time value using the account password as the secret key
  <Time />                      - UTC Unix time of the request for the TTL check and the HMAC hash authentication to the server
  <MList>                       - a collection of one or more API methods to be called
    <Meth>
      <MName />                 - the name of the API method being called
      <PList>                   - a collection of zero or more input parameters for the API method
        <Prm>                   - name/value pairs for each input parameter
          <Name />                          - the name of the input parameter for the API method
          <Val><![CDATA[ … ]]></Val>        - each input parameter value is encapsulated within a CDATA block
        </Prm>
      </PList>
    </Meth>
  </MList>
</ReqMsg>
```

The API response messages are limited to a maximum of 64K in size by the.NET XML reader in order to insure that the response messages will remain below the .NET 85,000 byte limit and not end up being stored in the LOH. Limiting the size of the response messages depends on the server-side pagination of tabular data.

API Response Message

```
<RespMsg>
  <UserName />                  - DGP system account name
  <ReqID />                     - unique ID created by the client app for each request message, and echoed back in the response
```

```
  <Time />                    - UTC Unix time of the response
  <Auth />                    - state of the request message authentication (OK, NoMatch, Expired, Disabled, Error, Exception)
  <Info />                    - optional information regarding Auth states other than OK
  <SvrMS />                   - the time spent on the server executing all of the API method calls in the request message batch
  <MethCount />               - the number of methods called in the request message batch
  <RList>                     - a variable collection of one or more API method results
    <Result>
      <RName />               - the name of the method result, used by the client to match results to method calls
      <RCode />               - code indicating the state of the method result (OK, Empty, Error, Exception)
      <DType />               - the data type of the result value (Int, Num, Text, DateTime, XML, JSON, DataTable)
      <RVal><![CDATA[ … ]]></Val>          - each return value is encapsulated within a CDATA block
    </Result>
  </RList>
</RespMsg>
```

XML fragments are used for the API request and response messages for the following reasons:

1. Cross Platform Ubiquity
   Structured text messages can be created and read by almost all of the most commonly used programming languages. In addition, almost all of these programming languages have the capability to work with XML already built in, and therefore do not require any 3rd party tools or SDK's in order to create API request messages or read API response messages.
2. CDATA Blocks
   All input parameter and method result values in API messages are enclosed within CDATA blocks, which eliminates the need to escape or encode the data they contain. This allows any type of data to be transported within the messages without breaking the structure of the messages themselves.
3. Memory Stream Reader

XML provides the ability to read messages as forward-scrolling cursors through a memory stream, eliminating the need for serialization/deserialization of the API messages. This mechanism is approximately 140 times faster than the fastest serialization/deserialization available. It is also the basis for the "tolerant reader" functionality needed by the API Test Harness and other DGP applications.

4. Security
   XML fragments eliminate the XML header, and can only contain data. All of the potential ways to exploit XML security vulnerabilities rely on the XML header (external entity attacks, expansion attacks, exploits for strong data types etc.). In addition, unlike JSON, it is not possible to inject executable instructions into XML.

Security

Client applications use TLS to encrypt and secure the API messages sent to and received from the web service API's. Another optional message encryption is available that mimics the hybrid cryptosystem used in PGP, but that only works for native applications.

In general, almost all functionality and data in a system are consolidated and collocated in the server tiers, so the client applications are only used to present data to the user and save input from the user. All security logic and most sensitive data stay on the servers and are not pushed out to each remote client application endpoint.

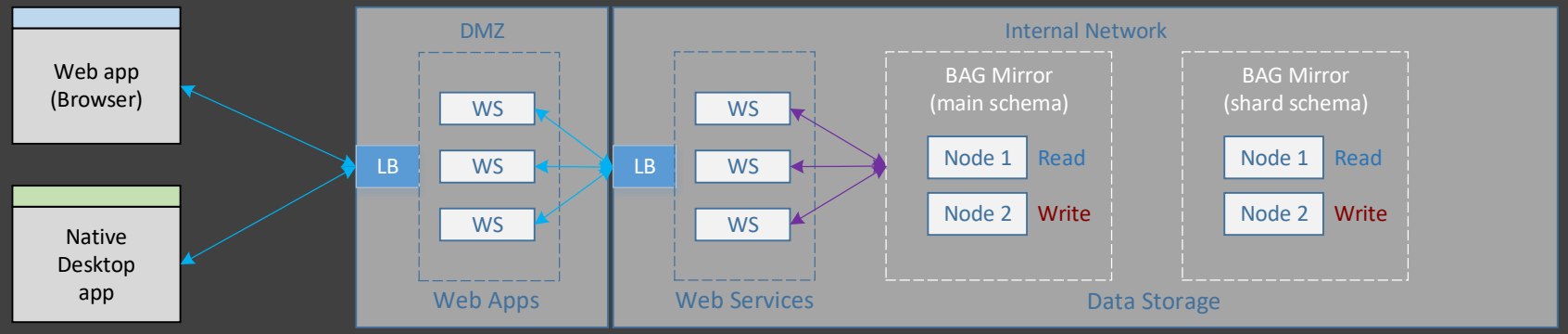
Message-based API's using a single-argument front controller limit the attack surface of each web service to a single parameter of a single controller method, which is the smallest external attack surface possible. TLS must be used to encrypt the API messages during transport. The TLS can be terminated either at the web server, or at the load balancer for larger locations that use them. Each web service then implements multiple layers of security and encryption (equivalent to an API security gateway) encapsulated within the message pipeline class used to process every API Request message.

A data-driven role-based access control (RBAC) custom identity store is used as the basis for both authentication and authorization of each individual API method called by an API request message. An HMAC hash mechanism is used for authentication of the API Request messages received by the server, and to authenticate the server to the client in the response of the Login API method. The Login method is also used to trigger lazy just-in-time updates of the user authorization data cached in each user account record in

order to improve overall system performance and scalability. Access to shared data in the FileStore app is authorized and controlled by data group membership and is transparently enforced within the data access methods.

Using open source projects in your custom software systems requires scanning the tree of open source projects for malicious code, recursively scanning the dependencies of each open source library and tool as well. The software required for these scans can be very expensive, and it is a lot of work to trace through all the dependencies of the libraries used by all the projects. DGP solves this problem by only using closed source Microsoft tools and frameworks for its own implementation (no open-source and no dependencies to external 3rd party libraries or tools). This eliminates the need to use any such scanning software in the first place.

Scalability and Availability



LB = Load Balancer | WS = Web Server | FS = File Server

Scaling the 4-tier architecture within a location is a matter of first separating each tier onto its own hardware, from which point each tier is able to scale independently of the others. The web server farms handle both N + 1 fault tolerance and incremental horizontal scalability for both the DMZ and internal network. The SQL Server Availability Groups provide redundancy for the various pairs of database servers. The main schema servers can only scale vertically. The shard schema servers provide horizontal scalability for shard data by incrementally adding more shard servers as needed.

Small systems will usually start out with a single computer for each of the 3 redundant locations. In this situation, each location must be taken offline sequentially for its monthly maintenance. Once a location has been scaled up to have its own redundancy with load-balanced web server farms and SQL Server Availability Groups, then the monthly maintenance can be performed within a location while it remains in use.

If the HYBFILE DGP schema is used, then the file storage shards may not be able to be maintained while they are in use, and the maintenance process would default back to the same mechanism used for small systems – taking entire locations offline sequentially for their monthly updates, patching, etc.

Performance

DGP has been designed and built to deliver high performance executables. This is necessary because of the end of Moore's Law, which means that performance improvements can no longer be accomplished by using newer, faster hardware to solve the problem of poor software system performance. Going forward, performance improvements can only be achieved by improving the architecture and code of the software systems themselves.

High performance is very beneficial to software systems in a number of ways. First, high performance helps provide an excellent user experience. Second, it enables the use of a server-centric architecture that consolidates all of the logic and data of a system in the middle tier, which improves the security of a system and greatly simplifies maintenance, debugging and deployments. However, this architecture is only feasible if the services of the middle tier are very fast. Third, the forced (and ongoing) simplification of a system's architecture and code that is needed to deliver high performance executables also results in a significant decrease in all types of resources used by the system, reducing both total costs and TTM.

The term "high performance" is defined as follows:

- Every user action in a UI, measured end-to-end, should be completed in under one second, for all of the different types of client applications. This means the application UI has not only displayed a new screen but that screen is also fully interactive in less than one second.

- To meet that requirement (especially on mobile devices), server-side processing must be completed in under 100 MS. DGP server processing on average is completed in approximately 10 MS.
- These are the minimum performance standards. Scalability can be increased, and total costs can be reduced even more with higher levels of performance and efficiency (which will eventually be subject to diminishing returns).

Building systems with high performance using the .NET Framework is accomplished by simplification of the architecture and code, along with improved memory management. Simplification equates to "removing complexity", which in practice means reducing the number of subsystems and components in the system architecture as much as possible. Removing these elements is achieved by eliminating the need for the subsystems in the first place. Beyond that step, further simplification comes down to finding ways for each remaining part of the architecture to serve more than one purpose simultaneously. In this way, a smaller number of parts is able to still meet the inherent level of functional complexity in a system. Logically, a DGP system has 3 tiers: Client, Web Service and Data Storage, with all the logic in a system consolidated into the web services. The only representation of the data in a system are the database schemas that are used to store the data, and the other tiers are designed to work directly with the database data in the form of DataTables (or XML structures if the metadata is removed). This extremely simple chain of IO-bound RPC's allows for the combination of thread pool threads and IO completion ports to use the available threads very efficiently.

Improved memory management using a managed-code framework means designing the implementation in such a way as to minimize the amount of GC work. The best summarized way to explain this is to write C# code as if it were Golang. Golang and Rust are both post-OO languages that use a mix of OO, functional and procedural coding styles to achieve high levels of simplicity and performance. C# code can be written the same way by carefully only using the simplest, lowest level parts of the .NET Framework, using minimal OO techniques to organize the code into as few classes and methods as logically possible, and then writing the majority of the functionality within the methods as procedural code.

Logging, Metrics and Monitoring

Logging is intended to capture data about events that have occurred in the remote application. This includes exceptions, errors and information. The basic pattern is to log data about the event locally to the Event Viewer, and then call an API method to log the same data to the DGPErrors table of the SysMetrics database. Tools would then monitor the DGPErrors table for the event data to be displayed in real-time dashboards, used for reporting and analytics, etc.

Metrics and monitoring takes a more proactive approach to collecting data about the system in an effort to verify that the system is running correctly and performing well. DGP monitoring consists of collecting end-to-end performance data from production systems under actual workloads during each day. Additional monitoring is actually a specialized type of testing used to collect data about important functionality within a system and save it to the LatticeMetrics table of the SysMetrics database.

Evolution

One of the main features of DGP web service API's is that they are able to frequently be extended, enhanced and otherwise improved over time in order to meet changing requirements. The primary rule in this process is that none of the improvements can ever break backward compatibility for all of the client apps and integrated systems that depend on the API functionality.

To meet this requirement, the web services follow an immutable append-only pattern for the API's, combined with explicitly named versions (ApiName.MethodName.VersionName). This allows old versions of API methods to coexist with newer versions, indefinitely. Client applications and integrated system are then able to migrate from old versions to new versions on their own timeline (if ever). Since existing methods are immutable once they have been deployed, that means that their test harness test files are also immutable. In practice, this means they require no maintenance work, so small teams of developers and testers only have to be able to keep up with the pace of the development of new features for the system over time.

The data-driven role-based access control (RBAC) security system provides fine-grained control over authorization to each version of each API method, and in addition acts as a feature toggle mechanism controlling access to all API functionality. Each environment has its own collection of databases, which means they each have their own security data, user accounts and roles, etc. Additions to the functionality of each environment can be rolled out in a controlled way to specific roles, as needed.

Universal Compatibility

The objective of a universal compatibility approach is to 1) allow any programming language to work with a single "generic" implementation of the web service API's, and 2) eliminate the need to develop and maintain client SDK's for each supported programming language.

Eliminating the need for client SDK's significantly reduces the amount of development and maintenance work for the provider of the API's, shortening delivery times and reducing their total cost. Also, this creates a very clear division between client application functionality and server-side API functionality. This allows the provider of the web service API's to only be responsible for the maintenance of their own server-side code and have no responsibility for any client application code whatsoever. This avoids potentially contentious support situations where the provider of the web services is responsible for the functionality of their SDK's that have been embedded within the code of the client application.

In order to achieve these objectives, a lowest-common-functionality approach is used for the API request and response messages, which are treated as formatted text messages, not as objects. They are not strongly typed, have no static structure, and therefore any of the major programming languages can work with them directly as text to create API request messages and read API response messages. The structure of the messages (XML fragments) is somewhat flexible, and uses a tolerant reader to allow additional elements to be appended in certain areas of the messages without breaking backward compatibility of the messages themselves.