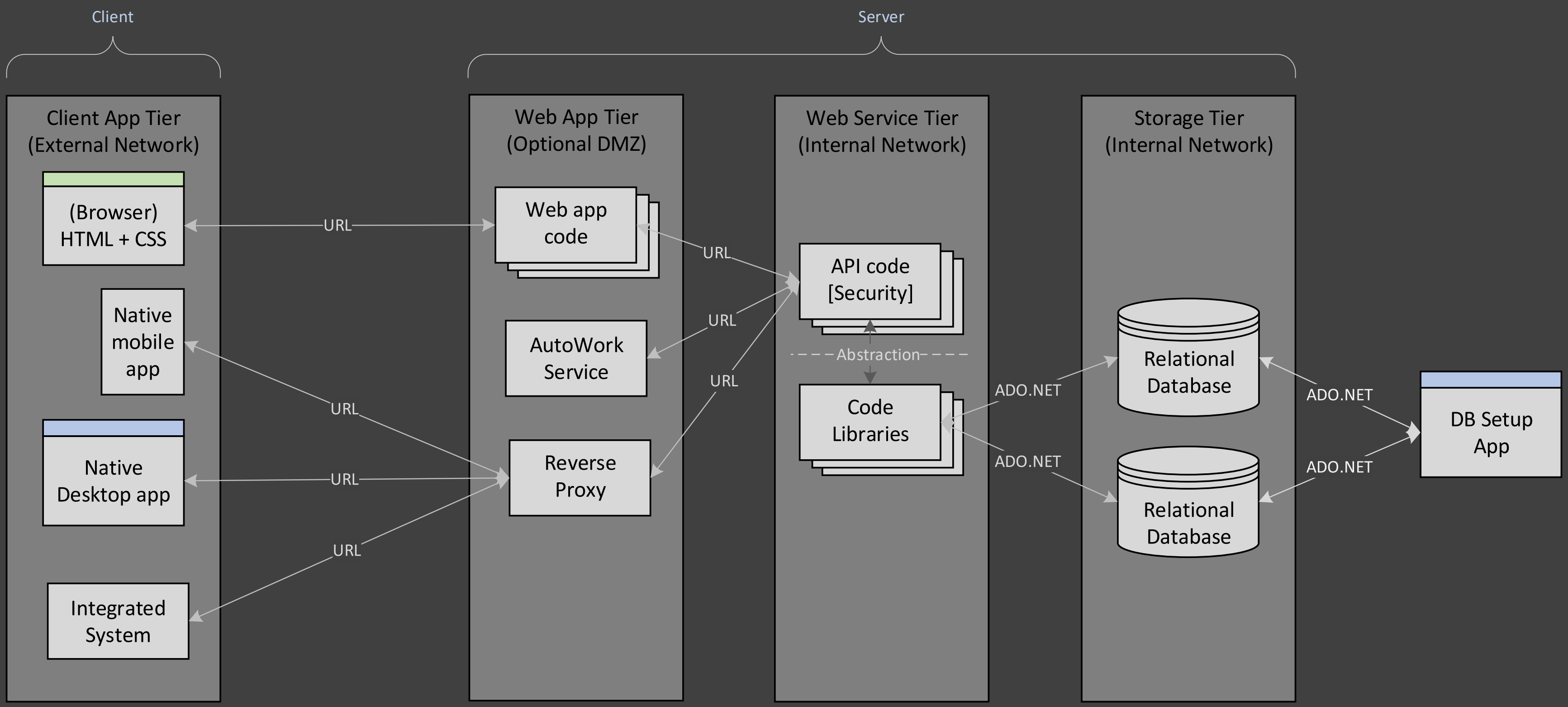
As much of the functionality as possible in a DGP system is consolidated into the web service API's. In practice, this means that the application tier contains very little logic, and the data storage tier contains none at all. This does not mean that applications or the database engine cannot contain logic – the logic in a system can be implemented in any of the tiers whenever that alternative produces the best overall results.



As shown in the diagram, the logic of a system is implemented as reusable code libraries behind the message-based web services. The web services are basically a mechanism to secure remote access to the functionality of the code libraries. This allows the code libraries to be used both remotely and also directly in the same memory space whenever possible. Calling the libraries in the same memory space is several hundred thousand times faster than an RPC. If situations arise where logic should only be called remotely, that logic can be built into the web services themselves.

There is an important layer of abstraction between the message-based API logic and the code libraries, which is basically the only layer of abstraction in the whole system. This one is very useful because it allows the external API's to be completely decoupled from the internal logic of the code libraries, which enables each to evolve independently from the other. Service evolution requires that the constant extensions, enhancements and improvements to a system never break backward compatibility. This is accomplished by following a strict immutable append-only pattern for the API method labels and their "signatures" of input parameters implemented in the mapper classes. Existing methods are considered immutable, and changes are added as new versions or new methods.

The consolidation of the majority of the logic into the web service API's simplifies and improves many aspects of the system:

1. Keeping all of the logic and data in the server tier helps to improve the overall security of the system by taking advantage of DGP's web service security architecture. Also, sensitive data never needs to leave the centralized, private and well protected server tier.
2. The API's are designed with end-to-end unit testing functionality and integrated test harness applications. The test harness allows both primary and alternate paths through the code to be tested. Altogether, this enables very close to 100% code coverage by the end-to-end tests of nearly all of the functionality in the entire system.
3. The majority of the processing work of a DGP system is handled by the processing nodes (web servers) running in load-balanced web server farms in each location. The load-balanced web server farm is the best physical infrastructure currently available. It provides the easiest and least expensive incremental horizontal scalability by simply adding additional servers to the farm as needed. At the same time it also enables the easiest and least expensive fault-tolerance by adding some ***extra*** servers to the farm to cover for any downtime among the other individual servers. Concentrating all of the logic into web service API's which are able to take advantage of the very best scalability and fault tolerance available maximizes those benefits for the overall system.
4. Reduced costs and delivery times
    a. It is much easier, less expensive, and takes less time to debug and fix centralized functionality compared to logic embedded and deployed into remote applications.
    b. It is much easier, less expensive, and takes less time to maintain and deploy (CI/CD/CT) centralized web services compared to installing/updating remote applications.

c. Thin client applications consist of only a presentation layer that calls the API's for all of their functionality, which makes the client applications much less expensive to develop, easier to test, and faster to deliver than client applications that contain most of the logic in a system.

The primary problem for this type of centralized functionality will usually be the slow responsiveness of applications that must call remote servers for everything they do. 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, which produce extremely fast web services that enable the use of a centralized architecture.

DGPWebServce Web.config

| Field Name | Field Values | Description |
|---|---|---|
| SvcKeyVersion | | The SvcKey label of current (latest) version of the system service key |
| SvcKey1 | 32-byte string (from GUID) | The encryption key value used for the service key version label. All versions are maintained in a collection of keys in the web.config file |
| | | |
| LocState | ONLINE, OFFLINE | A state value used to enable/disable the web service |
| System | | The name of the system the web service belongs to |
| Environment | | The environment name (dev, test, QA, prod) |
| Location | | The location name (descriptive) |
| WebSvcName | | The name of the web service, matching the IIS web app name |
| WebSvcVersion | YYYY-MM-DD | The current version of the web service being used, matching the date folder name of the web service |
| | | |
| EventSource | .NET Runtime | By default, the event source name is set to ".NET Runtime", which is an existing event source that is rarely used. If a custom event source can be created, then that name should be used instead. |

| | | |
|---|---|---|
| EventID | 1000 | An event ID (1000) that matches the default event source. An event ID is optional when a custom event source is used |
| TTLCheckFlag | ON, OFF | A flag value to turn the TTL check of API Request messages on or off |
| TTLMS | 10000 | The time in MS to be used by the TTL Check for API Request messages |
| UserCacheFlag | ON, OFF | A flag value to turn the caching of a user account UserInfo on or off |
| UserCacheSec | 600 | The number of seconds until the UserInfo in the Cache expires |
| RateLimitFlag | ON, OFF | A flag value to turn the method rate limit check per account on or off |
| MaxMethBatch | 10 | The maximum number of API methods that can be called by a single API Request message |
| MaxReqMsgKB | 64 | The maximum size of an API Request message in KB |
| MaxRespMsgKB | 64 | The maximum size of an API Response message in KB |
| MaxFailedLogin | 5 | The maximum number of failed login attempts allowed. Each successful login resets the count to zero |
| PasswordLength | 8 | The minimum length of new passwords (other password rules are enforced in the code |
| ExpireDays | 90 | The interval for setting new expiration dates, in days |
| MaxClaimBatch | 5 | The maximum number of records to be claimed by the API method called by the AutoWork scheduler |
| MaxFileSize | 200000000 | The maximum file size allowed to be stored by Lattice, in bytes |
| MaxSegSize | 45000 | The maximum file segment size (upload and download), in bytes |
| MaxFavorites | 100 | The maximum number of favorite files allowed in Lattice, per user |
| | | |
| FileStore | | The ADO.NET connection string for the FileStore database |
| FileShard1 | | The ADO.NET connection string for the first FileShard database |
| SysInfo | | The ADO.NET connection string for the SysInfo database |
| SysWork | | The ADO.NET connection string for the SysWork database |
| SysMetrics | | The ADO.NET connection string for the SysMetrics database |
| TestDB1 | | The ADO.NET connection string for the TestDB1 database |
| TestDB2 | | The ADO.NET connection string for the TestDB2 database |

## Security

Strong security is necessary to prevent a software system's data from being stolen or damaged, and also to prevent damage to the system itself. The consequences of poor security can result in extensive downtime, a ruined reputation or brand, lost sales, fines and lawsuits. Strong security is also a prerequisite for many certifications of compliance such as PCI-DSS, HIPPA, GDPR, etc. However, security is a cost center, so the challenge when designing software systems is to provide strong security at a relatively low cost, and without negatively impacting the productivity of the users. For DGP, this is accomplished by designing many reinforcing layers of security into the system's architecture from the ground up.

The overall context for DGP security is that 1) production networks are separate from the business network, 2) production locations have strong network perimeter security, and 3) TLS is used to encrypt traffic to and from each production location. To further strengthen security, DGP uses message-based web service API's and a single-argument front controller pattern to expose the minimum possible external attack surface (a single controller method that accepts a single input parameter).

The single method and single input parameter of each web service, along with size restrictions and lack of strong data types in API messages combine to effectively prevent buffer overflow attacks. In addition, DGP has been designed with the equivalent of an API security gateway integrated into each web service (the message pipeline class). This class is shared by all web services, and checks for the correct structure of each message, the message TTL, user account authentication, user account rate limit, and account authorization for each individual API method called in the request message batch.

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. The Login API method is 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 is authorized and controlled by data group membership and is transparently enforced within the data access methods.

The web services use no open-source libraries and have no dependencies to external 3rd party libraries or tools. Using open source projects in your custom software systems requires scanning the open source 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.

Also strengthening security are the 3 types of encryption used in DGP systems. Zero-knowledge encryption is used in the web services to encrypt data before it is stored in the database, and decrypt data retrieved from the database before it is returned to a client application. End-to-end encryption is used for all unstructured/file content in Lattice, with the client app encrypting files before they are uploaded to storage, and decrypting the files on the client after they have been downloaded. The encryption keys used in the end-to-end encryption are managed on the server and securely delivered to the client apps as needed. The final type of encryption is used to protect the secret values in the .config files using Microsoft's PKI, which is transparent to the web services and applications.

These multiple layers of security and encryption have successfully passed several PCI-DSS audits, and are just as effective at extremely small scales (such as a single computer) as they are in large distributed systems.

## Zero-knowledge Encryption

DGP zero knowledge encryption is used to encrypt sensitive data in the web service API's (using AES 256-bit symmetric encryption) and pass the encrypted data to be stored in its respective database table. To read the data the process is reversed, and the encrypted data is read from the database and decrypted in the web service API's before it is sent to the client application in an API response message.

The encryption keys are maintained in the Web.config file of the web services. Each encryption key used by the system over time has its own key in the .config file (and all encryption keys must be maintained in the Web.config file indefinitely for each DGP system). The key name of the current encryption key used to encrypt new data is stored as the SvcKeyVersion value. The key name of the encryption key used to encrypt data is stored in each database record along with the encrypted data. Decryption of the data in the web service uses the key name stored in each record to read the correct encryption key value from the Web.config file as needed.

```
    <!-- Service key values and current key label -->
    <add key="SvcKeyVersion" value="SvcKeyV1" />
    <add key="SvcKeyV1" value="abcdefghijklmnopqrstuvwxyz123456" />
```

## Performance Engineering

Another benefit to the consolidation and centralization of the logic is that it maximizes the benefits from efforts to significantly improve the performance and efficiency of the system. As mentioned elsewhere in the DGP documentation, 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 poorly performing software. Going forward, performance improvements can only be achieved by improving the architecture and code of the software systems themselves, and the majority of these improvements are concentrated in the centralized web services.

One of the best ways to describe the performance engineering techniques used in DGP is that everything has been built as if it were written using Golang. Like Rust, Golang is what comes after OO. It places a heavy emphasis on simplicity, and combines features of OO, procedural code and functional programming in order to significantly improve performance and efficiency. The .NET framework has those same capabilities, and it is therefore possible to achieve many of the performance improvements of Golang when using the .NET framework as a platform.

## Caching

The ASP.NET Cache object is used to cache some information regarding user sessions when using the web services. This stores the information in the memory of a web server with a setting for the amount of time until the cached value expires. It is used to store the decrypted UserInfo object, which eliminates the need for a query to the SysInfo identity store and the decryption of the account password. Also, the cache can be used to store the number of methods called by each account per minute to enforce rate limits. Since these values are stored in memory on a specific web server, session affinity is required when using a load balancer. If TLS is terminated on the web server instead of the load balancer, session affinity would basically be required anyway in order to reuse TLS connections once they have been established. This caching can be turned off using key values in the web.config file.

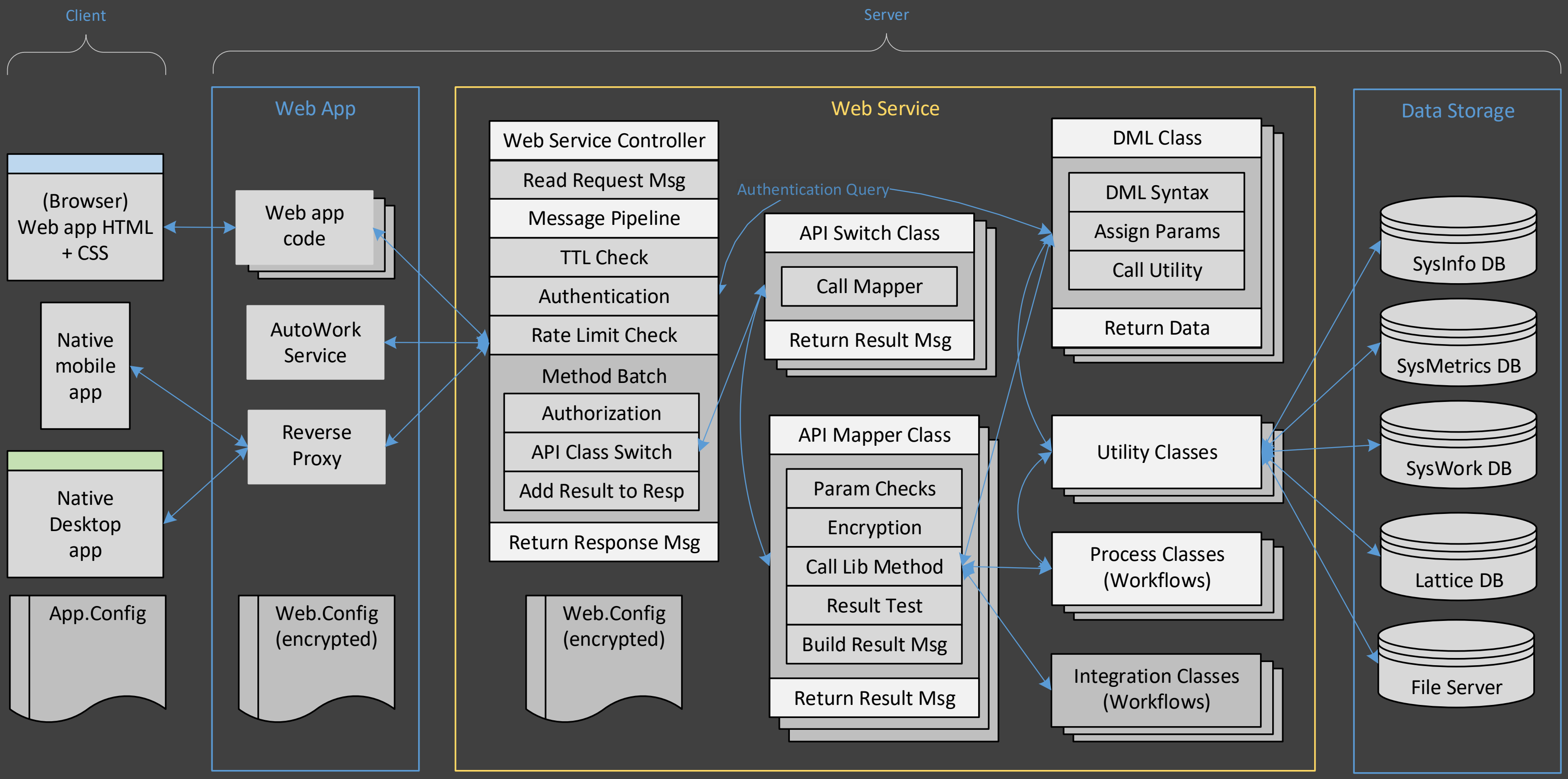
## Evolution

The system has been designed to allow for the functionality in a system to be constantly enhanced and expanded without ever breaking backward compatibility, in the least amount of time and the lowest cost. It does so by following an immutable append-only convention for all functionality in a system.

### Service Evolution

The system architecture only has one layer of abstraction, which exists between the web services and the shared libraries of code that do all of the actual work in a system. The web services, switch classes and mapper classes are the parts of a system that are accessible by remote applications and integrated systems, and therefore must follow the immutable append-only convention in order to avoid breaking backward compatibility.

The API methods that are called remotely are actually just text labels that are used to specify which internal method is being called by a client application or integrated system. These labels must follow the immutable append-only convention and can never be changed or deleted, while only new labels (API methods) can be added over time.



The API labels are used in switch classes to instantiate the correct Mapper classes. The mapper classes verify the API request message input parameters, instantiate the correct internal class, map the request message input parameters to their corresponding library method parameters, and call the internal method. These mapper methods must also follow the immutable append-only convention.

The internal libraries are completely decoupled from the web service API's and are therefore free to change to meet new requirements in whatever ways work best for them. The only rule is that the changes can never break the mapper methods that call them. This is verified by the test files of the API Tester test harness, which are used to run full regression tests of all API methods as part of every deployment to the dev, test and QA environments. The test files are actually running remote end-to-end tests of the mapper methods in each API. Once the mapper methods are deployed to a production environment, they become immutable, and the test files that test them also become immutable as well (bug fixes are the only exception to this rule).

Mapper method evolve either by adding optional parameters to the mapper method, with default values assigned when the optional parameters are missing, OR by adding a new version of a mapper method when new parameters are not optional. Either option requires a new test file to be created and added to the collection of test files. Most methods in a system are responsible for manipulating data in some way, and therefore have an informal dependency on the underlying data structure or schema.

When a new column is added to a database table in a DDL method, the change propagates upward into the code libraries and API code as well. If the new column is nullable, then optional parameters can be used in existing API methods and DML methods. If the new column requires a value, then existing methods must be modified to provide a default value, while new methods will accept a new input parameter to populate the new column. Default values can be added in the mapper methods (only used remotely), DML methods (used in the code libraries), or in the DDL method itself for “universal” default values.

The RBAC security is used as a feature toggle mechanism to control which groups of users (initially, only testers) are able to use new versions of methods in each environment.

## System Evolution

In the same way that new versions of methods can be added to an API and coexist with the older version(s), new versions of tiers can be added to a system and run in parallel with the old versions. This is made possible by the isolation between tiers, and the configurable endpoints of the RPC’s that are used to communicate between the tiers. Multiple versions of a tier are run in parallel long enough to allow for the gradual transition from an old version to a new version.

When done correctly, a transition to a new version of a tier can be done while the system itself remains in constant use.