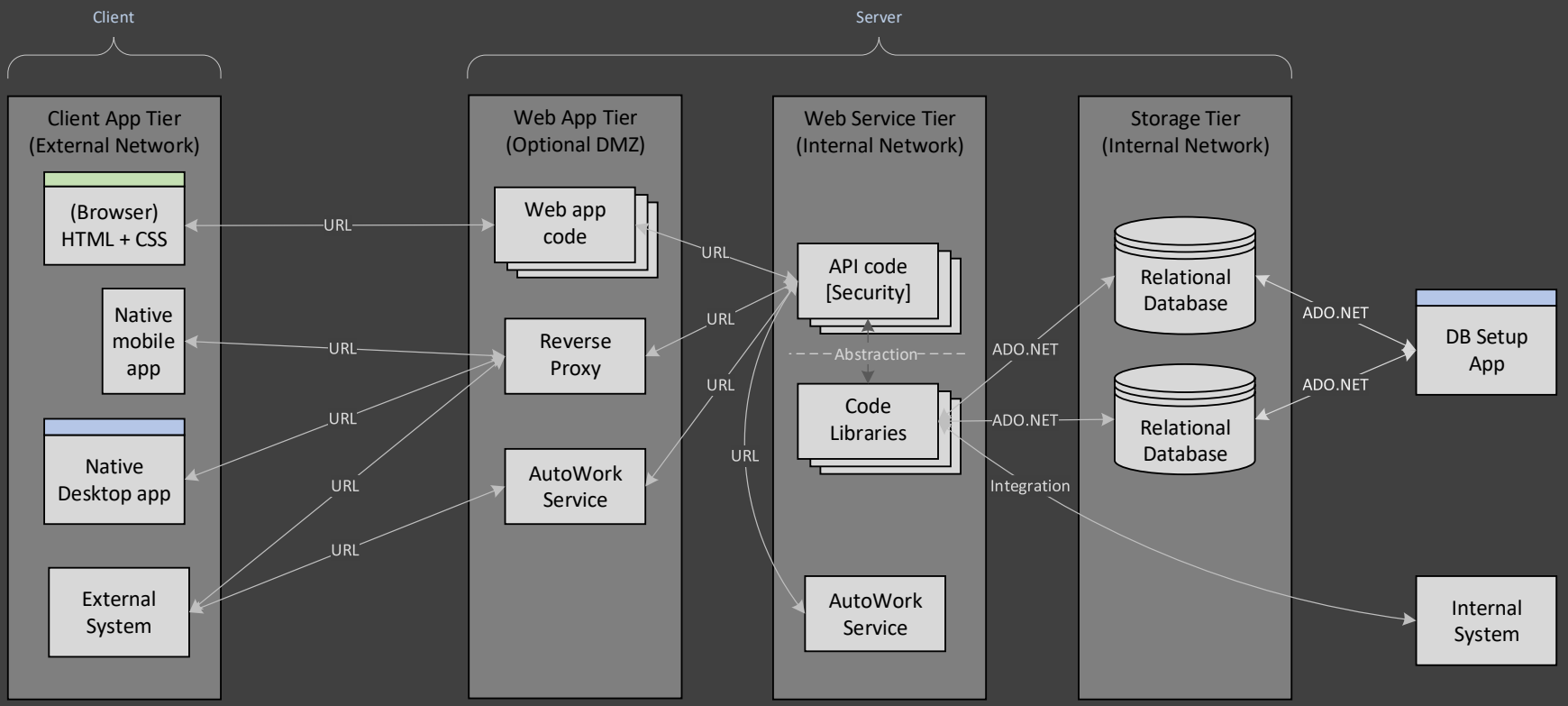
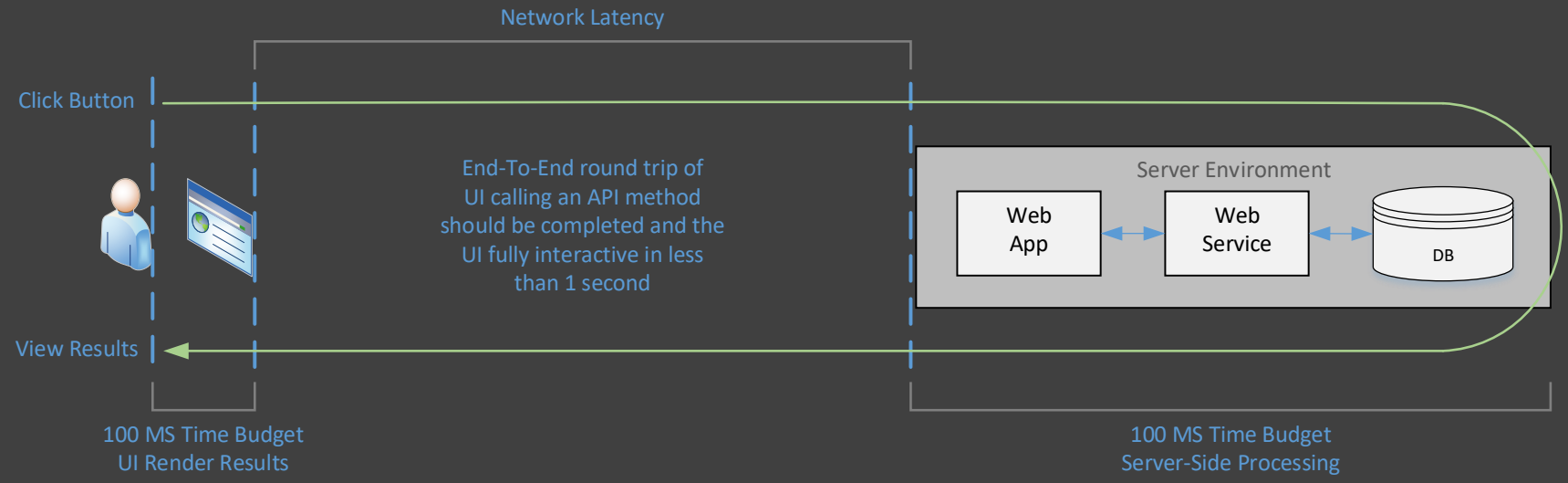
Modular Multi-tier Architecture



DGP uses a modular multi-tiered hub-and-spoke 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 run all of the tiers onto a single computer at the smallest scale. Systems will generally start out at a small scale in each distributed location and only add more processing nodes, storage nodes and infrastructure to each location if and when it is needed to scale up the system.

From a high level DGP deploys the hub-and-spoke architecture as a two-tier thin client/fat server topology which consolidates the system logic and data storage spokes into the server tier. Each tier is logically isolated from its adjacent tiers, and only communicate using various types of RPC. This isolation allows each tier to evolve and scale independently of its adjacent tiers, adding new features and functionality at its own pace and on its own timeline (which is also very important for system evolution). 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, which are automatically managed in Windows by IO completion ports – all of which is handled automatically by the .NET Framework CLR hosted within Windows. DGP simply makes use of these (and other) available capabilities when built to run on Windows, but the DGP architecture itself is not tied to Windows.

The primary problem for this type of centralized functionality will usually be the slow responsiveness of applications that must call remote servers as part of every UI action. In fact, this type of centralized hub-and-spoke architecture is only a viable option when the web service APIs are very fast. The **minimum** standard for UI performance is that *a UI must display the results of every user action and be fully interactive in less than one second.* A practical example would be: when a user clicks a button in the UI, the application 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 less than a second.



The reasons why one second is the important threshold are explained very well in this video from a Google engineer (Ilya Grigorik): "Speed, Performance, and Human Perception" https://www.youtube.com/watch?v=7ubJzEi3HuA . These performance standards are applicable to all types of applications, whether they are web, native, or mobile.

One second is the upper limit of the performance standard, but faster performance will further increase the scalability of a system, improve the user experience, and also further reduce costs. So overall, faster is better (subject to eventual diminishing returns).

Client Application Functionality

- Maintain User State (values cached temporarily in memory)
  - Endpoint URL's selected by the user
  - UserName and Password input by the user in the Connect screen
  - User account and service information returned by the Login method (method ACL, service public key, service state, etc.)
  - Pagination information per UI module

- Create API Request Messages
  - Create properly formatted API request messages (XML fragment)
  - (optional) Use the PGP-style hybrid cryptosystem to encrypt request messages and decrypt response messages
  - Use HTTP/HTTPS to POST API request messages to the web service API endpoint

- Security Token (SHA 256 hash used by default, HMAC hash is optional)
  - Create hash value security token from user credentials, which is used to authenticate the account
  - (optional) Verify HMAC hash values used to authenticate the Server to the Client App*

The API request messages are limited to a maximum of 80K in size in order to insure that the request messages will remain below the .NET 85,000 byte limit and avoid being stored in the LOH (Large Object Heap).

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>
```

- Read API Response Messages
    - Accept HTTP/HTTPS or SOAP response from the web service endpoint
    - Read properly formatted API response messages (XML fragment)

The API response messages are limited to a maximum of 83,000 bytes in size by the controller 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 also depends on the number of records set on the server to control 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>
```

- UI Feature Toggles
    - Application functionality enabled/disabled based on the web service version
    - Application functionality enabled/disabled based on user account role membership
    - Automated functionality enabled/disabled based on role membership, such as Remote Monitoring

In the UI, the effects of the feature toggles are applied when customizing the navigation menu. First, menu items are shown or hidden based on the role membership of the user account. Second, each section of the UI also has its own minimum web service

version (which is a date value). Certain sections of an application may require functionality from newer versions of the web service. If a user connects to an older web service that has not been updated, the sections that require a newer version would not be displayed even if the account was authorized to use them.

- 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

Optional Functionality

- Remote Monitoring
  - Performance metrics for all of the major sections of the UI are displayed in the status bar for users to observe for all environments of a system (especially useful in production environments).
  - For members of the RemoteMonitor role, the performance metrics (plus some other useful information) is saved to the SysMetrics table of the SysMetrics database.
  - DGPDrive users are able to verify the functionality of both logging to the local Event Viewer and logging to the centralized database table using the “Test Error Logging” menu item in the User menu.
  - Users that are members of the Testing role are able to run many end-to-end API test files that test and verify all of the functionality in a system. The test results can optionally be saved to the TestResult table of the SysMetrics database for future analysis.

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

1. Cross Platform Support
   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, and also helps to improve the security of the system. In addition, it is the basis for the "tolerant reader" functionality needed by the API Test Harness and other DGP applications.
4. Security
   XML can only contain data, and unlike JSON it is not possible to inject executable instructions into it. In addition, only XML fragments are used for DGP API request and response messages. Almost 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.).

Security

In general, virtually all functionality and data in a system are consolidated in the server tiers (which provides the best security by far), so the client applications are only used to present data to the user and collect new/edited data from the user. All security logic and most sensitive data stays on the servers and are never sent out to each remote client endpoint. Native apps are immutable compiled binaries and are therefore immune to many types of hacking that inject code into dynamic (interpreted) applications, such as those written in JavaScript for example. Web applications are not as secure in terms of immutability, but the use of Server Side Rendering (SSR) that keeps all logic and sensitive data on the server helps to significantly improve the security of web applications.

Scalability

The “scalability” of a UI mainly refers to its ability to work well and meet all of the performance requirements no matter how much data is stored in a system. This is made possible using two techniques. The first are mechanisms to only work with a small segment of the large sets of data at a time (pagination), and the second is good search functionality that is integrated to work with the server-side pagination mechanisms. The pagination of the data is mandated by the 80K limit to the size of API Request and Response messages enforced by the DGP web services. This is done both for performance and security reasons, and it also helps to prevent the creation of objects in the .NET Large Object Heap (LOH).

Some examples of this server-side pagination is the logic used for all of the main security tables in the DGPDrive Admin UI. The Search methods implement the pagination using SQL Server Offset Fetch statements. Combined with the Count methods using the same search criteria, the UI screens are able to move forward and backward through large amounts of data, one page at a time. The number of rows in the pages of data must be adjusted to avoid exceeding the 80K limit for API Request and Response messages.

A second example is the limitation on the maximum number of subfolders allowed in the FileStore application. The folder tree is built incrementally when the user clicks on a parent folder. The list of subfolders in this case is the equivalent of a page of data, and is subject to the same maximum size limitations.

Another type of scalability applies to the application itself. Using a modular architecture for the application allows additional functionality to be added easily over time (one part of system evolution). This modular architecture consists of a navigation system that loads subsections of an application into the main application screen. Again using DGPDrive as an example, the WinForm subsection screens are built as user controls which fill the main application screen when displayed by the navigation menus. New user controls can then be added to an application as needed, along with new navigation items to display those subsections. This modularity mimics the similar modularity of the reusable libraries of code that implement all of the work done by the system in the server tiers.

Performance

The standard for performance is that every user action in a UI must be displayed in a screen and the UI returned to a fully interactive state in less than one second. The performance of the client applications depends greatly on the performance of the web services, since that is the source of all the functionality and data in DGP systems. Native applications are deployed as binary executables, which helps improve their performance rendering screens once the data has been returned from the server. Web applications built using Server-Side Rendering (SSR) require time to call the appropriate web service API's and then build the finished HTML + CSS that incorporates the data from the server. This process should be completed by the web app in 100 MS or less to meet the 1 second overall performance requirement. By default, the browser only needs to render the pre-built HTML + CSS for each screen, which does not require much time. JavaScript libraries can be used to augment the functionality of the HTML + CSS as necessary, but they should be used in a way that does not interfere with the DOM and CSS pipelines in the browser, which otherwise can significantly delay the display of the page.

For both high performance and as the basis for the tolerant reader functionality, DGP applications read the API messages as one-way forward scrolling cursors through the HTTP request/response memory stream. This event-driven technique is roughly 140 times faster than the fastest serialization/deserialization implementations currently available. It is also the basis for the tolerant reader functionality which allows for the evolution of the API messages over time without breaking backward compatibility.

Availability

The availability of a system depends on the redundancy of its production environment server locations. By default, there will generally be 2 locations (a primary and a backup), separated geographically. Native applications use a system list file to keep track of the URL's used to connect to the web services in each of the locations. If a problem occurs with the location they are connected to, they would manually reconnect the native app to the web services in the other location.

Web applications are hosted in each location by default (although they can be hosted outside of a location as needed). If a problem occurs with the location they are using, the user would manually switch their browser to the URL of the other location.

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.

API Version Dependencies

The Login method returns the version date of the web service to which it has connected. This version date can be used as a feature toggle in the client application to enable or disable functionality that depends on newer versions of the API's when connecting to older versions of the web services.

The DGPDrive application is an example of this functionality. It receives the list of user account roles in the response from the Login method, and enables sections of the navigation menus accordingly. The logic to use the role membership as feature toggles also compares the version date of the web service to the minimum date values needed to support that section of the UI, and if the web service is older than that date, it will disable the section even if the user role membership would allow it to be used.

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 SDKs 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 serialized objects. They are also not strongly typed, have no static structure, and therefore any of the major programming languages can work with them directly as structured 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.