

Google API Client: Proposed Authentication Design

Keep one `TGoogleAPIClient`, introduce interchangeable user and service-account authenticators, and let the common authenticator base also handle bearer-token request signing. The API services continue using the same request infrastructure regardless of the authentication method.

This follows the **Strategy pattern** described in the supplied suggestions: the client depends on a common authentication interface, while separate implementations obtain tokens in different ways. [1] The combined design adds an FPC-specific simplification: the authenticator also implements the existing request-signer interface, so a separate bearer-signer object is unnecessary.

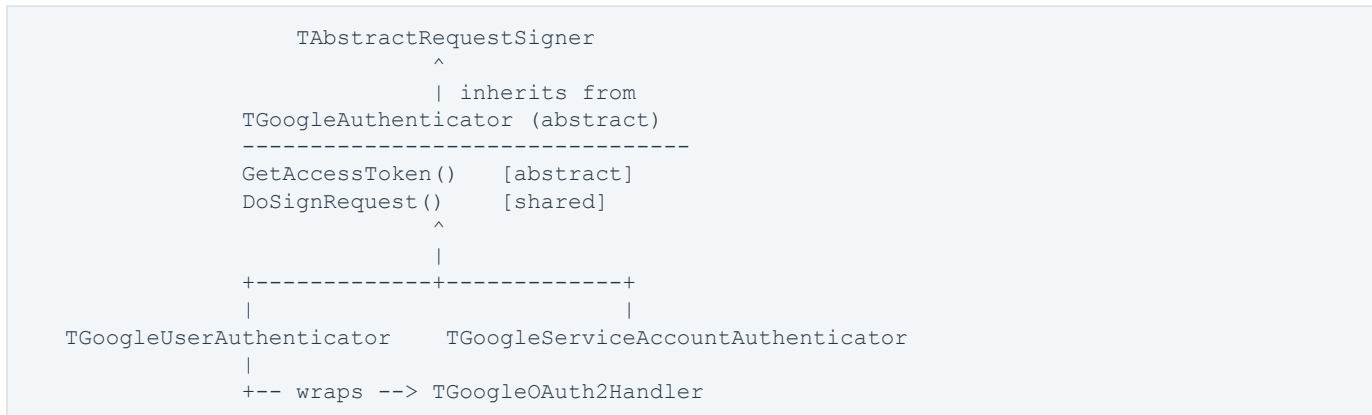
1. Summary of architectural changes

Area	Proposed change
Main client	Retain <code>TGoogleAPIClient</code> as the application-facing client and service configurator.
Authentication dependency	Add an <code>Authenticator</code> property typed as the abstract <code>TGoogleAuthenticator</code> , rather than requiring every authentication path to use <code>TGoogleOAuth2Handler</code> .
User authentication	Add <code>TGoogleUserAuthenticator</code> , which wraps the existing user OAuth implementation.
Service-account authentication	Add <code>TGoogleServiceAccountAuthenticator</code> , which independently implements signed-JWT token acquisition.
Request signing	Make <code>TGoogleAuthenticator</code> inherit from <code>TAbstractRequestSigner</code> and implement bearer-header handling once in the base class.
Common token interface	Require only <code>GetAccessToken</code> for normal token acquisition. It returns a usable cached token or obtains a replacement.
Browser authorization	Keep interactive authorization explicit and specific to the user authenticator.
Existing API services	Preserve their <code>WebClient</code> and <code>RequestSigner</code> integration; do not introduce another HTTP request wrapper.

The selected authenticator determines the authentication method. **A separate writable `AuthMode` property is not needed.**

2. Class hierarchy

The two authentication implementations share a base class but do not inherit from each other.



TGoogleAuthenticator

Defines the common contract and implements shared request signing.

Its `GetAccessToken` method is abstract. Each concrete authenticator supplies the appropriate token-acquisition behavior. This is the small interface proposed in the `TGoogleAuthenticator` document. [1]

Its shared `DoSignRequest` method:

1. Calls `GetAccessToken`.
2. Rejects an empty or malformed token.
3. Sets `Authorization: Bearer <access token>`.

If authentication fails, an exception stops the request before it is sent.

TGoogleUserAuthenticator

Adapts the existing `TGoogleOAuth2Handler` to the common interface. The supplied design likewise proposes wrapping the existing user OAuth implementation and keeping user-authorization operations on this subclass. [1]

The wrapper should reuse the existing handler's token and session state rather than maintain a second, competing token cache. It exposes explicit authorization operations, such as `AuthorizeUser`, and obtains replacement access tokens through the existing user-token renewal flow.

TGoogleServiceAccountAuthenticator

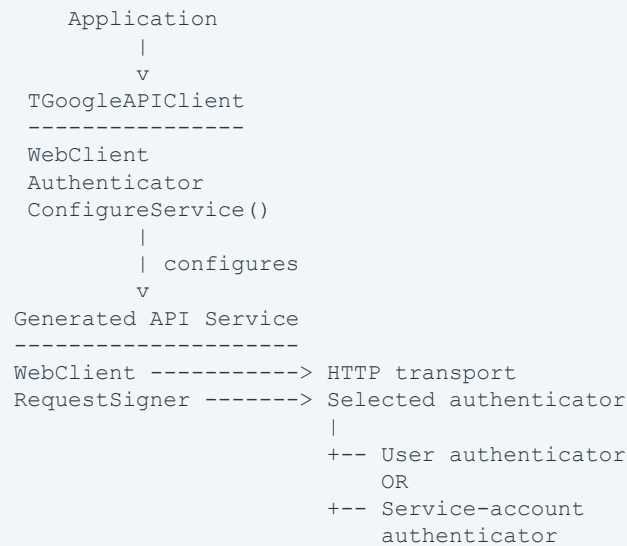
Loads service-account credentials and manages its own access-token state. When a replacement token is needed, it creates a signed JWT assertion and exchanges that assertion for an OAuth access token. It does not use a refresh token. [1]

The proposed configuration includes service-account identity, signing credentials, scopes, and an optional delegated `Subject` for the previously discussed Workspace impersonation case.

The signed JWT assertion is used to obtain an access token; the resulting access token is what the shared signer attaches to API requests.

3. Client and service relationships

TGoogleAPIClient connects an API service to the HTTP transport and the selected authenticator.



The essential service configuration remains:

```

AService.WebClient := FWebClient;
AService.RequestSigner := FAuthenticator;

```

These assignments illustrate the proposed integration, not a complete implementation.

The generated service does not inspect credential files, select authentication flows, open browsers, or sign JWT assertions. It delegates request signing to the configured authenticator.

Use one client instance per credential context. Applications needing both authentication methods simultaneously create separate client instances with different authenticators.

4. Request and token flow

Both implementations follow the same request path:



Only the replacement-token step differs:

Behavior	User authenticator	Service-account authenticator
Initial authorization	Explicit user consent and authorization-code exchange	Programmatic signed-JWT assertion exchange
Routine token renewal	Uses the saved user refresh token	Creates and exchanges another signed assertion
Browser interaction during ordinary API calls	None; reports that user authorization is required when necessary	Never
Token state	Reuses the existing OAuth handler's session state	Maintains its own token and expiry
Header added to API requests	Authorization: Bearer <access token>	Authorization: Bearer <access token>

The user-refresh and service-account assertion-exchange paths are the two token-acquisition strategies described in the supplied examples. [1]

Token acquisition must use an **unsigned token-endpoint request**, outside the API bearer-signing path. Otherwise, requesting a token could recursively invoke the signer that is trying to obtain it.

5. Configuration, compatibility, and lifecycle

Credential selection. Provide explicit loaders such as `LoadUserCredentials` and `LoadServiceAccountCredentials`. A convenience `LoadCredentials` method may detect supported JSON formats and construct the appropriate authenticator. Each concrete class validates its own configuration.

Backward compatibility. Preserve existing user-specific properties and the legacy `AuthHandler` interface as compatibility accessors to the user implementation. Keep service-account settings separate; do not repurpose `ClientSecret` or `RefreshToken` to hold private keys or JWT assertions.

Object lifetime. Define ownership clearly. Authenticators created by the client can be client-owned; externally supplied authenticators need a documented ownership policy. Configure the authenticator before configuring services, and do not replace or free it while those services are using it. Because services reference the authenticator directly as their signer, replacing it requires reconfiguring those services.

Token-state boundaries. Cache tokens with expiry information and a renewal margin. Changes to identity, scopes, delegated subject, or signing credentials must not reuse the previous context's token. Synchronize renewal when supporting concurrent requests.

Result

The design has three distinct responsibilities:

```
TGoogleAPIClient
    Configures and connects the objects.

TGoogleAuthenticator
    Obtains an access token and applies the shared bearer header.

Concrete authenticator
    Implements either user OAuth or service-account token acquisition.
```

The API request path stays unchanged, the existing user OAuth implementation is reused, and service-account support is added without forcing it into a user-specific authentication hierarchy.

Source note

[1] Supplied Gemini architecture suggestion: user and service account for google apis.docx. Referenced sections: Strategy Design Pattern; Define an Authentication Strategy Base; Implement the User Credentials Strategy (3-Legged); Implement the Service Account Strategy (2-Legged).