+++ title = "Add authentication for data source plugins" aliases = ["/docs/grafana/latest/plugins/developing/auth-for-datasources/", "/docs/grafana/next/developers/plugins/authentication/"] +++

Add authentication for data source plugins

This page explains how to configure your data source plugin to authenticate against a third-party API.

There are two ways you can perform authenticated requests from your plugin—using the *data source proxy*, or by building a *backend plugin*. The one you choose depends on how your plugin authenticates against the third-party API.

- Use the data source proxy if you need to authenticate using Basic Auth or API keys
- Use the data source proxy if the API supports OAuth 2.0 using client credentials
- Use a backend plugin if the API uses a custom authentication method that isn't supported by the data source proxy, or if your API communicates over a different protocol than HTTP

Regardless of which approach you use, you first need to encrypt any sensitive information that the plugin needs to store.

Encrypt data source configuration

Data source plugins have two ways of storing custom configuration: `jsonData` and `secureJsonData`.

Users with the *Viewer* role can access data source configuration—such as the contents of `jsonData`—in cleartext. If you've enabled anonymous access, anyone that can access Grafana in their browser can see the contents of `jsonData`. **Only use `jsonData` to store non-sensitive configuration.**

> ***Note:*** *You can see the settings that the current user has access to by entering `window.grafanaBootData` in the developer console of your browser.*

> ***Note:*** *Users of Grafana Enterprise can restrict access to data sources to specific users and teams. For more information, refer to Data source permissions.*

If you need to store sensitive information, such as passwords, tokens and API keys, use `secureJsonData` instead. Whenever the user saves the data source configuration, the secrets in `secureJsonData` are sent to the Grafana server and encrypted before they're stored.

Once the secure configuration has been encrypted, it can no longer be accessed from the browser. The only way to access secrets after they've been saved is by using the *data source proxy*.

Add secret configuration to your data source plugin

To demonstrate how you can add secrets to a data source plugin, let's add support for configuring an API key.

Create a new interface in `types.go` to hold the API key.

```
export interface MySecureJsonData {
  apiKey?: string;
}
```

Add type information to your `secureJsonData` object by updating the props for your `ConfigEditor` to accept the interface as a second type parameter.

```
interface Props extends DataSourcePluginOptionsEditorProps<MyDataSourceOptions,
MySecureJsonData> {}
```

You can access the value of the secret from the `options` prop inside your `ConfigEditor` until the user saves the configuration. When the user saves the configuration, Grafana clears the value. After that, you can use the `secureJsonFields` to determine whether the property has been configured.

```
const { secureJsonData, secureJsonFields } = options;
const { apiKey } = secureJsonData;
```

To securely update the secret in your plugin's configuration editor, update the `secureJsonData` object using the `onOptionsChange` prop.

```
const onAPIKeyChange = (event: ChangeEvent<HTMLInputElement>) => {
  onOptionsChange({
    ...options,
    secureJsonData: {
      apiKey: event.target.value,
    },
  });
};
```

Next, define a component that can accept user input.

```
<Input
  type="password"
  placeholder={secureJsonFields?.apiKey ? 'configured' : ''}
  value={secureJsonData.apiKey ?? ''}
  onChange={onAPIKeyChange}
/>
```

Finally, if you want the user to be able to reset the API key, then you need to set the property to `false` in the `secureJsonFields` object.

```
const onResetAPIKey = () => {
  onOptionsChange({
    ...options,
    secureJsonFields: {
      ...options.secureJsonFields,
      apiKey: false,
    },
    secureJsonData: {
      ...options.secureJsonData,
      apiKey: '',
    },
  });
};
```

Now that users can configure secrets, the next step is to see how we can add them to our requests.

Authenticate using the data source proxy

Once the user has saved the configuration for a data source, any secret data source configuration will no longer be available in the browser. Encrypted secrets can only be accessed on the server. So how do you add them to you request?

The Grafana server comes with a proxy that lets you define templates for your requests. We call them *proxy routes*. Grafana sends the proxy route to the server, decrypts the secrets along with other configuration, and adds them to the request before sending it off.

> ***Note:*** *Be sure not to confuse the data source proxy with the [auth proxy]({{< relref "../../auth/auth-proxy.md" >}}). The data source proxy is used to authenticate a data source, while the auth proxy is used to log into Grafana itself.*

Add a proxy route to your plugin

To forward requests through the Grafana proxy, you need to configure one or more proxy routes. A proxy route is a template for any outgoing request that is handled by the proxy. You can configure proxy routes in the plugin.json file.

1. Add the route to plugin.json. Note that you need to restart the Grafana server every time you make a change to your plugin.json file.

```
"routes": [
  {
    "path": "example",
    "url": "https://api.example.com"
  }
]
```

2. In the `DataSource`, extract the proxy URL from `instanceSettings` to a class property called `url`.

```
export class DataSource extends DataSourceApi<MyQuery, MyDataSourceOptions> {
  url?: string;

  constructor(instanceSettings:
DataSourceInstanceSettings<MyDataSourceOptions>) {
    super(instanceSettings);

    this.url = instanceSettings.url;
  }

  // ...
}
```

3. In the `query` method, make a request using `BackendSrv`. The first section of the URL path needs to match the `path` of your proxy route. The data source proxy replaces `this.url + routePath` with the `url` of the route. The following request will be made to `https://api.example.com/v1/users`.

```
import { getBackendSrv } from '@grafana/runtime';
```

```
const routePath = '/example';

getBackendSrv().datasourceRequest({
  url: this.url + routePath + '/v1/users',
  method: 'GET',
});
```

Add a dynamic proxy route to your plugin

Grafana sends the proxy route to the server, where the data source proxy decrypts any sensitive data and interpolates the template variables with the decrypted data before making the request.

To add user-defined configuration to your routes, add `{{ .JsonData.apiKey }}` to the route, where `apiKey` is the name of a property in the `jsonData` object.

```
"routes": [
  {
    "path": "example",
    "url": "https://api.example.com/projects/{{ .JsonData.projectId }}"
  }
]
```

You can also configure your route to use sensitive data by using `.SecureJsonData`.

```
"routes": [
  {
    "path": "example",
    "url": "https://{{ .JsonData.username }}:{{ .SecureJsonData.password
}}@api.example.com"
  }
]
```

In addition to the URL, you can also add headers, URL parameters, and a request body, to a proxy route.

Add HTTP headers to a proxy route

```
"routes": [
  {
    "path": "example",
    "url": "https://api.example.com",
    "headers": [
      {
        "name": "Authorization",
        "content": "Bearer {{ .SecureJsonData.apiToken }}"
      }
    ]
  }
]
```

Add URL parameters to a proxy route

```
"routes": [
  {
    "path": "example",
    "url": "http://api.example.com",
    "urlParams": [
      {
        "name": "apiKey",
        "content": "{{ .SecureJsonData.apiKey }}"
      }
    ]
  }
]
```

Add a request body to a proxy route

```
"routes": [
  {
    "path": "example",
    "url": "http://api.example.com",
    "body": {
      "username": "{{ .JsonData.username }}",
      "password": "{{ .SecureJsonData.password }}"
    }
  }
]
```

Add a OAuth 2.0 proxy route to your plugin

The data source proxy supports OAuth 2.0 authentication.

Since the request to each route is made server-side, only machine-to-machine authentication is supported. In order words, if you need to use a different grant than client credentials, you need to implement it yourself.

To authenticate using OAuth 2.0, add a `tokenAuth` object to the proxy route definition. If necessary, Grafana performs a request to the URL defined in `tokenAuth` to retrieve a token before making the request to the URL in your proxy route. Grafana automatically renews the token when it expires.

Any parameters defined in `tokenAuth.params` are encoded as `application/x-www-form-urlencoded` and sent to the token URL.

```
{
  "routes": [
    {
      "path": "api",
      "url": "https://api.example.com/v1",
      "tokenAuth": {
        "url": "https://api.example.com/v1/oauth/token",
        "params": {
```

```
          "grant_type": "client_credentials",
          "client_id": "{{ .SecureJsonData.clientId }}",
          "client_secret": "{{ .SecureJsonData.clientSecret }}"
        }
      }
    }
  ]
}
```

Authenticate using a backend plugin

While the data source proxy supports the most common authentication methods for HTTP APIs, using proxy routes has a few limitations:

- Proxy routes only support HTTP or HTTPS
- Proxy routes don't support custom token authentication

If any of these limitations apply to your plugin, you need to add a [backend plugin]({{< relref "./backend/_index.md" >}}). Since backend plugins run on the server they can access decrypted secrets, which makes it easier to implement custom authentication methods.

The decrypted secrets are available from the `DecryptedSecureJSONData` field in the instance settings.

```
func (ds *dataSource) QueryData(ctx context.Context, req *backend.QueryDataRequest)
(*backend.QueryDataResponse, error) {
  instanceSettings := req.PluginContext.DataSourceInstanceSettings

  if apiKey, exists := settings.DecryptedSecureJSONData["apiKey"]; exists {
    // Use the decrypted API key.
  }

  // ...
}
```

Forward OAuth identity for the logged-in user

If your data source uses the same OAuth provider as Grafana itself, for example using [Generic OAuth Authentication]({{< relref "../../auth/generic-oauth.md" >}}), your data source plugin can reuse the access token for the logged-in Grafana user.

To allow Grafana to pass the access token to the plugin, update the data source configuration and set the `jsonData.oauthPassThru` property to `true`. The DataSourceHttpSettings provides a toggle, the **Forward OAuth Identity** option, for this. You can also build an appropriate toggle to set `jsonData.oauthPassThru` in your data source configuration page UI.

When configured, Grafana will pass the user's token to the plugin in an Authorization header, available on the `QueryDataRequest` object on the `QueryData` request in your backend data source.

```
func (ds *dataSource) QueryData(ctx context.Context, req *backend.QueryDataRequest)
(*backend.QueryDataResponse, error) {
  token := strings.Fields(req.Headers["Authorization"])
```

```
    var (
        tokenType   = token[0]
        accessToken = token[1]
    )

    for _, q := range req.Queries {
        // ...
    }
}
```

In addition, if the user's token includes an ID token, Grafana will pass the user's ID token to the plugin in an `X-ID-Token` header, available on the `QueryDataRequest` object on the `QueryData` request in your backend data source.

```
func (ds *dataSource) QueryData(ctx context.Context, req *backend.QueryDataRequest)
(*backend.QueryDataResponse, error) {
  idToken := req.Headers["X-ID-Token"]

    for _, q := range req.Queries {
        // ...
    }
}
```

The `Authorization` and `X-ID-Token` headers will also be available on the `CallResourceRequest` object on the `CallResource` request in your backend data source when `jsonData.oauthPassThru` is `true`.

```
func (ds *dataSource) CallResource(ctx context.Context, req
*backend.CallResourceRequest, sender backend.CallResourceResponseSender) error {
  token := req.Headers["Authorization"]
  idToken := req.Headers["X-ID-Token"] // present if user's token includes an ID
token

  // ...
}
```

> ***Note:*** *Due to a bug in Grafana, using this feature with PostgreSQL can cause a deadlock. For more information, refer to Grafana causes deadlocks in PostgreSQL, while trying to refresh users token.*