+++ title = "Add support for variables in plugins" +++

Add support for variables in plugins

Variables are placeholders for values, and can be used to create things like templated queries and dashboard or panel links. For more information on variables, refer to [Templates and variables]({{< relref "../../variables/_index.md" >}}).

This guide explains how to leverage template variables in your panel plugins and data source plugins.

We'll see how you can turn a string like this:

```
SELECT * FROM services WHERE id = "$service"
```

into

```
SELECT * FROM services WHERE id = "auth-api"
```

Grafana provides a couple of helper functions to interpolate variables in a string template. Let's see how you can use them in your plugin.

Interpolate variables in panel plugins

For panels, the `replaceVariables` function is available in the PanelProps.

Add `replaceVariables` to the argument list, and pass it a user-defined template string.

```
export const SimplePanel: React.FC<Props> = ({ options, data, width, height,
replaceVariables }) => {
  const query = replaceVariables('Now displaying $service');

  return <div>{query}</div>;
};
```

Interpolate variables in data source plugins

For data sources, you need to use the getTemplateSrv, which returns an instance of TemplateSrv.

1. Import `getTemplateSrv` from the `runtime` package.

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

2. In your `query` method, call the `replace` method with a user-defined template string.

   ```
   async query(options: DataQueryRequest<MyQuery>): Promise<DataQueryResponse> {
     const query = getTemplateSrv().replace('SELECT * FROM services WHERE id =
   "$service"', options.scopedVars);

     const data = makeDbQuery(query);
   ```

```
  return { data };
}
```

Format multi-value variables

When a user selects multiple values for variable, the value of the interpolated variable depends on the variable format.

A data source can define the default format option when no format is specified by adding a third argument to the interpolation function.

Let's change the SQL query to use CSV format by default:

```
getTemplateSrv().replace('SELECT * FROM services WHERE id IN ($service)',
options.scopedVars, 'csv');
```

Now, when users write `$service`, the query looks like this:

```
SELECT * FROM services WHERE id IN (admin,auth,billing)
```

For more information on the available variable formats, refer to [Advanced variable format options]({{< relref "../../variables/advanced-variable-format-options.md" >}}).

Set a variable from your plugin

Not only can you read the value of a variable, you can also update the variable from your plugin. Use `locationService.partial(query, replace)`.

The following example shows how to update a variable called `service`.

- `query` contains the query parameters you want to update. Query parameters controlling variables are prefixed with `var-`.
- `replace: true` tells Grafana to update the current URL state, rather than creating a new history entry.

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

```
locationService.partial({ 'var-service': 'billing' }, true);
```

> ***Note:*** *Grafana queries your data source whenever you update a variable. Excessive updates to variables can slow down Grafana and lead to a poor user experience.*

Add support for query variables to your data source

[Query variables]({{< relref "../../variables/variable-types/add-query-variable.md" >}}) is a type of variable that allows you to query a data source for the values. By adding support for query variables to your data source plugin, users can create dynamic dashboards based on data from your data source.

Let's start by defining a query model for the variable query.

```
export interface MyVariableQuery {
  namespace: string;
  rawQuery: string;
}
```

For a data source to support query variables, you must override the `metricFindQuery` in your `DataSourceApi` class. `metricFindQuery` returns an array of `MetricFindValue` which has a single property, `text`:

```
async metricFindQuery(query: MyVariableQuery, options?: any) {
  // Retrieve DataQueryResponse based on query.
  const response = await this.fetchMetricNames(query.namespace, query.rawQuery);

  // Convert query results to a MetricFindValue[]
  const values = response.data.map(frame => ({ text: frame.name }));

  return values;
}
```

> ***Note:*** *By default, Grafana provides a default query model and editor for simple text queries. If that's all you need, then you can leave the query type as* `string`.
>
> ```
> async metricFindQuery(query: string, options?: any)
> ```

Let's create a custom query editor to allow the user to edit the query model.

1. Create a `VariableQueryEditor` component.

```
import React, { useState } from 'react';
import { MyVariableQuery } from './types';

interface VariableQueryProps {
  query: MyVariableQuery;
  onChange: (query: MyVariableQuery, definition: string) => void;
}

export const VariableQueryEditor: React.FC<VariableQueryProps> = ({ onChange,
query }) => {
  const [state, setState] = useState(query);

  const saveQuery = () => {
    onChange(state, `${state.query} (${state.namespace})`);
  };

  const handleChange = (event: React.FormEvent<HTMLInputElement>) =>
    setState({
      ...state,
      [event.currentTarget.name]: event.currentTarget.value,
    });
```

```
  return (
    <>
      <div className="gf-form">
        <span className="gf-form-label width-10">Namespace</span>
        <input
          name="namespace"
          className="gf-form-input"
          onBlur={saveQuery}
          onChange={handleChange}
          value={state.namespace}
        />
      </div>
      <div className="gf-form">
        <span className="gf-form-label width-10">Query</span>
        <input
          name="rawQuery"
          className="gf-form-input"
          onBlur={saveQuery}
          onChange={handleChange}
          value={state.rawQuery}
        />
      </div>
    </>
  );
};
```

Grafana saves the query model whenever one of the text fields loses focus (`onBlur`) and then previews the values returned by `metricFindQuery` .

The second argument to `onChange` allows you to set a text representation of the query which will appear next to the name of the variable in the variables list.

2. Finally, configure your plugin to use the query editor.

```
import { VariableQueryEditor } from './VariableQueryEditor';

export const plugin = new DataSourcePlugin<DataSource, MyQuery,
MyDataSourceOptions>(DataSource)
  .setQueryEditor(QueryEditor)
  .setVariableQueryEditor(VariableQueryEditor);
```

That's it! You can now try out the plugin by adding a [query variable]({{< relref "../../variables/variable-types/add-query-variable.md" >}}) to your dashboard.