+++ title = “Working with data frames” +++

Working with data frames

The data frame is a columnar data structure which allows efficient querying of large amounts of data. Since data frames are a central concept when developing plugins for Grafana, in this guide we’ll look at some ways you can use them.

The DataFrame interface contains a `name` and an array of `fields` where each field contains the name, type, and the values for the field.

> **Note:** If you’re looking to migrate an existing plugin to use the data frame format, refer to [Migrate to data frames]({{< relref “migration-guide.md#migrate-to-data-frames” >}}).

Create a data frame

If you build a data source plugin, then you’ll most likely want to convert a response from an external API to a data frame. Let’s look at how to create a data frame.

Let’s start with creating a simple data frame that represents a time series. The easiest way to create a data frame is to use the toDataFrame function.

```
// Need to be of the same length.
const timeValues = [1599471973065, 1599471975729];
const numberValues = [12.3, 28.6];

// Create data frame from values.
const frame = toDataFrame({
  name: 'http_requests_total',
  fields: [
    { name: 'Time', type: FieldType.time, values: timeValues },
    { name: 'Value', type: FieldType.number, values: numberValues },
  ],
});
```

> **Note:** Data frames representing time series contain at least a `time` field, and a `number` field. By convention, built-in plugins use `Time` and `Value` as field names for data frames containing time series data.

As you can see from the example, creating data frames like this requires that your data is already stored as columnar data. If you already have the records in the form of an array of objects, then you can pass it to `toDataFrame` which tries to guess the schema based on the types and names of the objects in the array. If you’re creating complex data frames this way, then be sure to verify that you get the schema you expect.

```
const series = [
  { Time: 1599471973065, Value: 12.3 },
  { Time: 1599471975729, Value: 28.6 },
];

const frame = toDataFrame(series);
frame.name = 'http_requests_total';
```

Read values from a data frame

When you're building a panel plugin, the data frames returned by the data source are available from the `data` prop in your panel component.

```
const SimplePanel: React.FC<Props> = ({ data }) => {
  const frame = data.series[0];

  // ...
};
```

Before you start reading the data, think about what data you expect. For example, to visualize a time series we'd need at least one time field, and one number field.

```
const timeField = frame.fields.find((field) => field.type === FieldType.time);
const valueField = frame.fields.find((field) => field.type === FieldType.number);
```

Other types of visualizations might need multiple dimensions. For example, a bubble chart that uses three numeric fields: the X-axis, Y-axis, and one for the radius of each bubble. In this case, instead of hard coding the field names, we recommend that you let the user choose the field to use for each dimension.

```
const x = frame.fields.find((field) => field.name === xField);
const y = frame.fields.find((field) => field.name === yField);
const size = frame.fields.find((field) => field.name === sizeField);

for (let i = 0; i < frame.length; i++) {
  const row = [x?.values.get(i), y?.values.get(i), size?.values.get(i)];

  // ...
}
```

Alternatively, you can use the DataFrameView, which gives you an array of objects that contain a property for each field in the frame.

```
const view = new DataFrameView(frame);

view.forEach((row) => {
  console.log(row[options.xField], row[options.yField], row[options.sizeField]);
});
```

Display values from a data frame

Field options let the user control how Grafana displays the data in a data frame.

To apply the field options to a value, use the `display` method on the corresponding field. The result contains information such as the color and suffix to use when display the value.

```
const valueField = frame.fields.find((field) => field.type === FieldType.number);

return (
  <div>
    {valueField
      ? valueField.values.toArray().map((value) => {
          const displayValue = valueField.display!(value);
          return (
            <p style={{ color: displayValue.color }}>
              {displayValue.text} {displayValue.suffix ? displayValue.suffix : ''}
            </p>
          );
        })
      : null}
  </div>
);
```

To apply field options to the name of a field, use getFieldDisplayName.

```
const valueField = frame.fields.find((field) => field.type === FieldType.number);
const valueFieldName = getFieldDisplayName(valueField, frame);
```