+++ title = “Build a streaming data source plugin” +++

Build a streaming data source plugin

This guide explains how to build a streaming data source plugin.

This guide assumes that you’re already familiar with how to [Build a data source plugin]({{< relref “/tutorials/build-a-data-source-plugin.md” >}}).

When monitoring critical applications, you want your dashboard to refresh as soon as your data does. In Grafana, you can set your dashboards to automatically refresh at a certain interval, no matter what data source you use. Unfortunately, this means that your queries are requesting all the data to be sent again, regardless of whether the data has actually changed.

By enabling *streaming* for your data source plugin, you can update your dashboard as soon as new data becomes available.

For example, a streaming data source plugin can connect to a websocket, or subscribe to a message bus, and update the visualization whenever a new message is available.

Let’s see how you can add streaming to an existing data source!

Grafana uses RxJS to continuously send data from a data source to a panel visualization. There’s a lot more to RxJS than what’s covered in this guide. If you want to learn more, check out the RxJS documentation.

1. Enable streaming for your data source in the `plugin.json` file.

   ```
   {
     "streaming": true
   }
   ```

2. Change the signature of the `query` method to return an `Observable` from the `rxjs` package. Make sure you remove the `async` keyword.

   ```
   import { Observable } from 'rxjs';

   query(options: DataQueryRequest<MyQuery>): Observable<DataQueryResponse> {
     // ...
   }
   ```

3. Create an `Observable` for each query, and then combine them all using the `merge` function from the `rxjs` package.

   ```
   import { Observable, merge } from 'rxjs';

   const observables = options.targets.map((target) => {
     return new Observable<DataQueryResponse>((subscriber) => {
       // ...
     });
   ```

```
});

return merge(...observables);
```

4. In the `subscribe` function, create a `CircularDataFrame`.

```
import { CircularDataFrame } from '@grafana/data';

const frame = new CircularDataFrame({
  append: 'tail',
  capacity: 1000,
});

frame.refId = query.refId;
frame.addField({ name: 'time', type: FieldType.time });
frame.addField({ name: 'value', type: FieldType.number });
```

   Circular data frames have a limited capacity. When a circular data frame reaches its capacity, the oldest data point is removed.

5. Use `subscriber.next()` to send the updated data frame whenever you receive new updates.

```
import { LoadingState } from '@grafana/data';

const intervalId = setInterval(() => {
  frame.add({ time: Date.now(), value: Math.random() });

  subscriber.next({
    data: [frame],
    key: query.refId,
    state: LoadingState.Streaming,
  });
}, 500);

return () => {
  clearInterval(intervalId);
};
```

   Note: In practice, you'd call `subscriber.next` as soon as you receive new data from a websocket or a message bus. The example above simulates data being received every 500 milliseconds.

Here's the final `query` method.

```
query(options: DataQueryRequest<MyQuery>): Observable<DataQueryResponse> {
  const streams = options.targets.map(target => {
    const query = defaults(target, defaultQuery);

    return new Observable<DataQueryResponse>(subscriber => {
      const frame = new CircularDataFrame({
```

```
        append: 'tail',
        capacity: 1000,
      });

      frame.refId = query.refId;
      frame.addField({ name: 'time', type: FieldType.time });
      frame.addField({ name: 'value', type: FieldType.number });

      const intervalId = setInterval(() => {
        frame.add({ time: Date.now(), value: Math.random() });

        subscriber.next({
          data: [frame],
          key: query.refId,
          state: LoadingState.Streaming,
        });
      }, 100);

      return () => {
        clearInterval(intervalId);
      };
    });
  });

  return merge(...streams);
}
```

One limitation with this example is that the panel visualization is cleared every time you update the dashboard. If you have access to historical data, you can add, or *backfill*, it to the data frame before the first call to `subscriber.next()`.