Develop with FastTrack

Generated on: 11 September 2025

Copyright

Table of Contents

Chapter 1: Introduction to FastTrack 6

Architecture of a FastTrack enabled React application 7

Chapter 2: Set up a three-tiered application 8

Data Tier: MarkLogic 9

Middle Tier: Node.js Express 11

UI Tier: React 12

Chapter 3: Create a search application 13

Requirements 13

Install FastTrack 13

Connect the application to MarkLogic 14

Add the FastTrack UI widgets 15

Search MarkLogic using MarkLogicContext 15

Example rendering 16

Build a search application with other FastTrack widgets 17

Chapter 4: Referencing values in data with PathConfig 18

Chapter 5: Include document extracts in search results 23

Chapter 6: Add a search result popup window 27

Example document 27
Handle clicks and display results 28

Chapter 7: Add faceted search to an application 32
Existing search application 32
Example document 32
MarkLogic setup 33
Configure a constraint in the query options 34
Add the StringFacet widget 36
Rendered result 38

Chapter 8: Add typehead search for SearchBox 39

Chapter 9: FastTrack scenarios 42
Scenario 1: High-value Target 42
Scenario 2: Crime Map 42

Chapter 10: FastTrack widgets 44
Avatar 44
BucketRangeFacet 47
CategoricalChart 54
CommentBox 74
CommentList 78
ChatBot 84
DataGrid 90
DateRangeFacet 95
EntityRecord 102

ExportData 107
GeoMap. 110
JsonView 121
MarkLogicContext 124
NetworkGraph 154
NumberRangeFacet 173
ResultsSnippet 180
ResultsConfig 183
ResultsCustom 188
SearchBox. 192
SelectedFacets 196
Slider 201
StringFacet 203
SemanticQuery 208
Timeline. 213
TwoLayersChart 227
WindowCard 242

1

Introduction to FastTrack

FastTrack is a library of search, visualization, and analytics widgets used to build React applications quickly on top of MarkLogic's powerful multi-model database and search platform. FastTrack can be imported into a React application. The widgets can then be connected to MarkLogic's backend APIs, and data retrieved from MarkLogic displayed in a variety of ways.

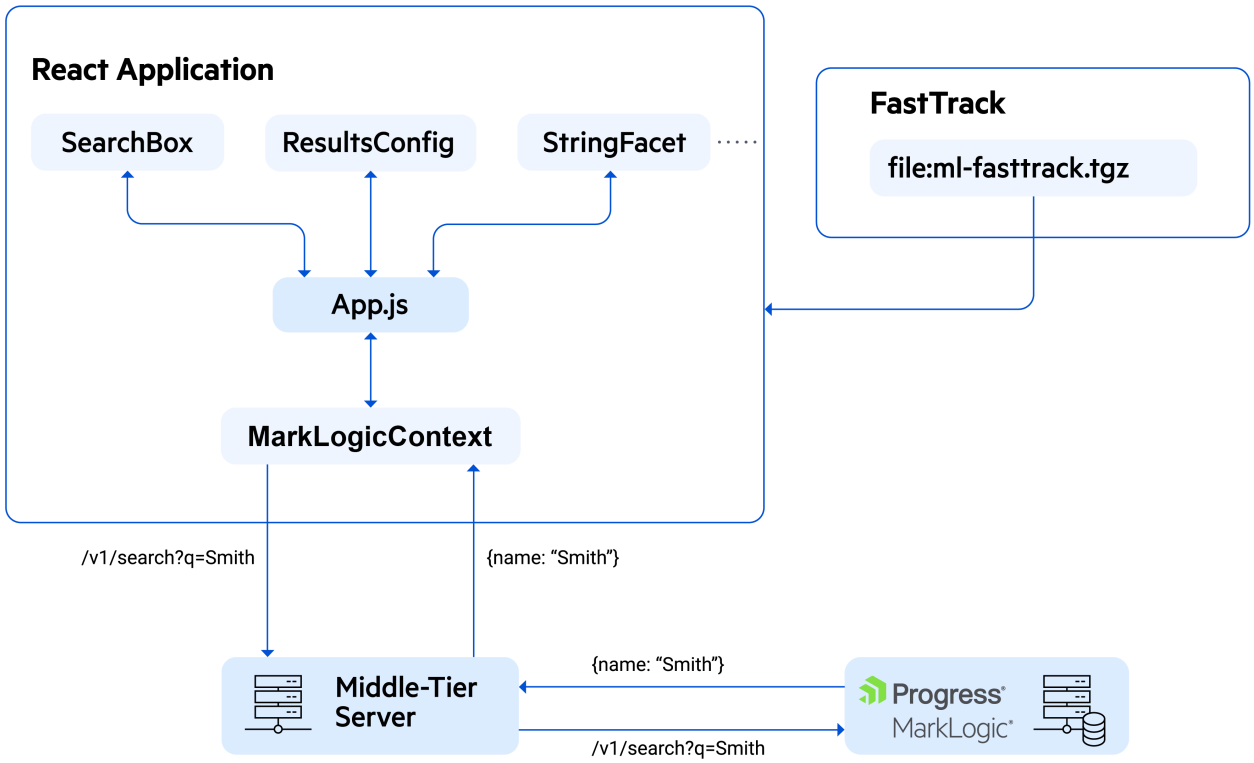
FastTrack lets you build:

- Faceted search applications (SearchBox, ResultsSnippet, ResultsConfig, ResultsCustom, StringFacet, BucketRangeFacet, NumberRangeFacet, DateRangeFacet).
- Network graphs for visualizing relationships in data (NetworkGraph).
- Timelines for organizing time-based information (Timeline).
- Maps for displaying geospatial data and constraining that data with polygons (GeoMap).
- Dashboards with interactive charts (CategoricalChart, TwoLayersChart).

FastTrack's MarkLogicContext connects your application to MarkLogic and exposes methods for communicating with MarkLogic's REST endpoints. MarkLogicContext lets you add interactivity across a React application by defining MarkLogic search constraints, executing complex structured searches that include those constraints, and updating application widgets as new search results are returned.

Architecture of a FastTrack enabled React application

FastTrack widgets allow users to visualize and interact with MarkLogic data. Communication between the application and MarkLogic is brokered by `MarkLogicContext`.



2

Set up a three-tiered application

These steps set up a three-tier application that can use FastTrack to search MarkLogic data.

> **Note:**
>
> You can set up a three-tier application using the `fasttrack-getting-started.zip` file included with FastTrack.

1. Data tier: MarkLogic server with data exposed through a REST API via an app server.
2. Middle tier: Node.js Express server that forwards requests from the UI to MarkLogic, and responses from MarkLogic to the UI.
3. UI tier: React application that can import and use the FastTrack widgets.

After setting up the three-tier application, a search application can be created to experiment with the FastTrack widgets. See Create a search application.

> **Note:**
>
> Two-tier applications can be built with a UI that communicates directly with MarkLogic. However, this may sacrifice the flexibility and security advantages of using a middle tier.

Data Tier: MarkLogic

The steps in this section configure MarkLogic for a FastTrack-enabled project. Complete these steps with MarkLogic server installed and running.

Databases and App Server

MarkLogic databases store the documents that a FastTrack project searches. A MarkLogic app server exposes these documents through a REST API. These setup steps use the Documents and Modules databases and the App-Services app server. These are automatically created when MarkLogic is initialized.

> **Note:**
>
> - The setup script configures the App-Services app server to use basic authentication.
> - The Documents and Modules databases are located under the **Databases** heading in the MarkLogic Administration Interface. App-Services is located under **App Servers**. The Administration Interface can be accessed on port 8001 (http://localhost:8001/).

User

MarkLogic applications require a user with permission to execute searches via MarkLogic's REST API. The set up script creates a new `fasttrack-getting-started-user` with a `rest-admin` role.

For more information about MarkLogic users and roles, see the *Security Guide*.

Documents

The set up script loads a set of JSON documents that describe persons in the Documents database. These documents can be searched with the application. Here is an example of a document:

```
{ "envelope": {
    "entityType": "person",
```

```
    "id": 1001,
    "firstName": "Nerta",
    "lastName": "Hallwood",
    "title": "Marketing Manager",
    "status": "active",
    "dob": "1985-03-04",
    "salary": 104000,
    "address": {
      "street": "40 Summer Ridge Point",
      "city": "Cincinnati",
      "state": "Ohio",
      "country": "United States",
      "latitude": 39.1848,
      "longitude": -84.3448
    },
    "image": "person-1001.jpg",
    "content": "And this, our life, exempt from public haunt, finds tongues
in trees, books in the running brooks, sermons in stones, and good in
everything.",
    "relations": [
      {
        "triple": {
          "subject": "http://example.org/1001",
          "predicate": "http://xmlns.com/foaf/0.1/knows/",
          "object": {
              "datatype" : "http://www.w3.org/2001/XMLSchema#string",
              "value":"1002"
            }
          }
      },
      {
        "triple": {
          "subject": "http://example.org/1001",
          "predicate": "http://xmlns.com/foaf/0.1/knows/",
          "object": {
              "datatype" : "http://www.w3.org/2001/XMLSchema#string",
              "value":"1003"
            }
          }
      }
    ]
  }
}
```

Query options

Query options can customize MarkLogic search results. The set up script loads query options into the Modules database that allow a match "snippet" to be returned for each result. A snippet includes contextual information about what values in the documents matched the search query.

For more information about query options and how they affect search results, see Search Customization Using Query Options.

Install dependencies

Install these dependencies by running this command from the `fasttrack-getting-started/setup` directory:

```
npm install
```

Run the set up script

To configure the app server, set up a user, load example documents, and install query options:

1. Run this command from the `fasttrack-getting-started/setup` directory:

   ```
   npm start
   ```

   The set up script configuration is stored in the `setup/config.js` file. This file can be edited if needed.

2. After running the command, confirm that documents were loaded by opening MarkLogic's QConsole application at port 8000 and explore the Documents and Modules databases at http://localhost:8000/qconsole.

Middle Tier: Node.js Express

The example application communicates with MarkLogic through a Node.js Express server (the architecture's middle tier). The middle tier passes search requests to MarkLogic, returns search responses from MarkLogic, and applies HTTP headers to search requests to avoid any CORS obstacles. In a real-world application, the middle tier would also commonly handle authentication tasks. The UI will communicate with the middle tier through port 4001.

1. Install the middle-tier dependencies by running this command from the `fasttrack-getting-started/server` directory:

   ```
   npm install
   ```

2. Start the middle-tier server by running this command:

   ```
   npm start
   ```

3. The middle-tier server gets its configuration information from a file at `server/config.js`. This file can be edited if necessary.

UI Tier: React

FastTrack widgets are React components that function inside a React application. This project includes a React UI application in the `my-app` directory that was scaffolded using Vite. FastTrack widgets can be added to the React UI to enable MarkLogic search.

To start the React application:

1. In a new terminal, change into the `my-app` directory and install the application dependencies with this command:

   ```
   npm install
   ```

2. Start the React UI application by running this command:

   ```
   npm start
   ```

3. Once the React application is running, it can be accessed at http://localhost:3000.

Now that the three-tiered application is set up, the FastTrack library can be installed, and the widgets can be used to build a MarkLogic search application. See Create a search application.

3

Create a search application

This section will explain how to use FastTrack widgets to build a React application that searches MarkLogic and displays the search results.

Requirements

To use FastTrack widgets to build a search application, these items are required:

- Node.js 18+
- npm 9+
- MarkLogic 11+
- Middle-tier server

Install FastTrack

Note:

These steps assume:

- Documents are loaded into a MarkLogic database and an app server that expose the MarkLogic REST endpoints.
- A middle-tier server proxies MarkLogic REST requests and handles authentication and CORS. See Set up a three-tiered application for setup instructions.

To install FastTrack:

1. Scaffold your React application and copy the `ml-fasttrack-x.x.x.tgz` file to the application root directory. A React application can be scaffolded using a tool such as Vite.
2. Run this command from the application root directory in a terminal:

```
npm install file:ml-fasttrack-x.x.x.tgz
```

Connect the application to MarkLogic

After installing FastTrack, connect the application to MarkLogic by wrapping the application with the `MarkLogicProvider` widget and add props for the middle-tier server. `MarkLogicProvider` provides access to `MarkLogicContext`, which exposes methods for calling MarkLogic's REST endpoints. Replace the code in the `main.jsx` file with this:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import { MarkLogicProvider } from "ml-fasttrack"

ReactDOM.createRoot(document.getElementById('root')).render(
  <MarkLogicProvider
    scheme="http"
    host="localhost"
    port="4001"
    options="search-options"
    auth={{ username: "fasttrack-getting-started-user", password: "password"
}}
  >
    <App />
  </MarkLogicProvider>
)
```

> **Note:**
>
> This application passes a user name and password with each request. The middle tier passes that information to MarkLogic. In a real-world application, authentication would commonly be handled in the middle tier.

Add the FastTrack UI widgets

Next, add the FastTrack widgets. In this example, the `MarkLogicContext`, `ResultsSnippet`, and `SearchBox` widgets are added. Replace the code in the `App.jsx` file with this:

```
import React from 'react'
import './App.css'
import { useContext } from "react";
import { MarkLogicContext, SearchBox, ResultsSnippet } from "ml-fasttrack"

function App() {
  const context = useContext(MarkLogicContext);
  return (
    <div>
      <SearchBox/>
      <ResultsSnippet/>
    </div>
  )
}

export default App
```

Widget definitions

- `MarkLogicContext` - communicates with MarkLogic.
- `ResultsSnippet` - displays search results.
- `SearchBox` - allows a user to enter and submit search text.

Search MarkLogic using MarkLogicContext

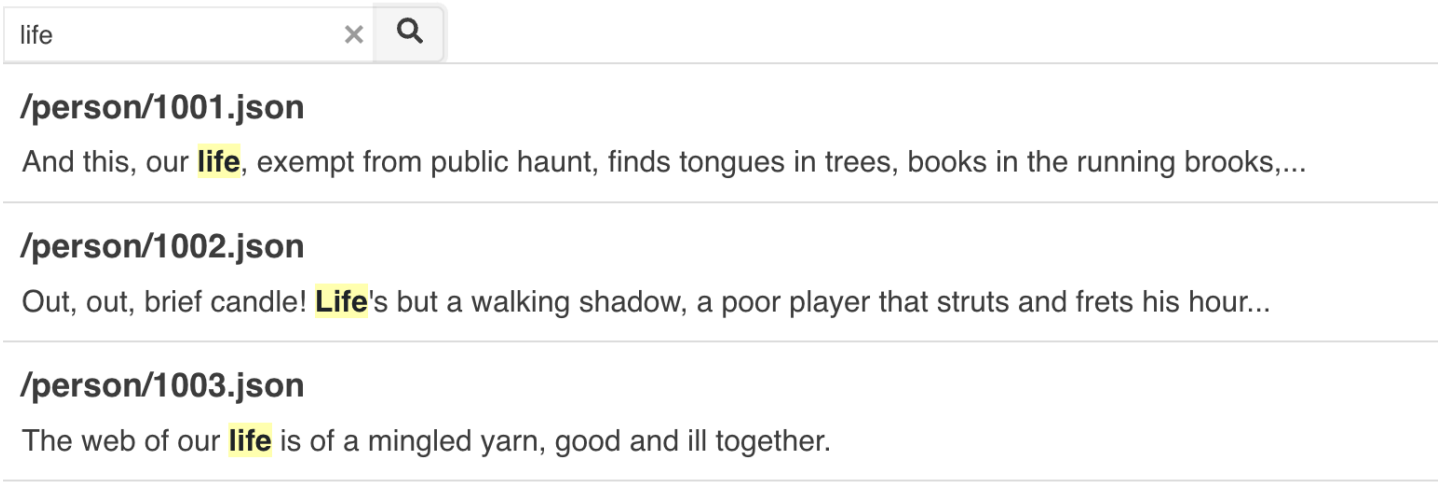
When a search is submitted, this code uses the `onSearch` event handler from `SearchBox` to set the search text in `MarkLogicContext`. `MarkLogicContext` recognizes a change in state, executes a search, and

stores the result in the state variable `searchResponse`. The results are displayed with `ResultsSnippet`. Replace the code in the `App.jsx` file with this:

```
import React from 'react'
import './App.css'
import { useContext } from "react";
import { MarkLogicContext, SearchBox, ResultsSnippet } from "ml-fasttrack"

function App() {
  const context = useContext(MarkLogicContext);
  return (
    <div>
      <SearchBox
        onSearch={(params) => context.setQtext(params.q)}
      />
      <ResultsSnippet
        results={context.searchResponse.results}
      />
    </div>
  )
}

export default App
```

Example rendering

The code in Search MarkLogic using MarkLogicContext renders this screen:



Build a search application with other FastTrack widgets

Search applications can be built using other FastTrack widgets by following the same pattern:

1. Add widgets from FastTrack to the application.
2. Use event handlers from the widgets to execute searches using `MarkLogicContext`.
3. Display the search results stored in the `MarkLogicContext` state (or from the payloads returned from `MarkLogicContext`).

4

Referencing values in data with PathConfig

FastTrack widgets use a common method for referencing values in data -- a PathConfig object. In its simplest form, PathConfig includes a `path` property that references a value in the data using a JSONPath string. FastTrack uses the jsonpath-plus library as its underlying JSONPath implementation.

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}} // Data
  config={{ path: "result.prop2" }} // PathConfig object that references a
value ("logic") in the data
/>
```

In this example, the `config` prop takes a `PathConfig` object. The `path` value in the `PathConfig` object references `"Logic"` in the `data` object.

An alternative to using the `path` property is using the `paths` property (for referencing multiple values) and the `value` property (for providing a static value). The referenced value(s) can be transformed by setting additional properties in the `PathConfig` object. See the API table and examples for the available properties and how the transformations are applied.

Path Config API

PathConfig Property Name	Type	Example	Description
path	string	`"result.prop2"`	A JSONPath string that references a value.
paths	string[]	`[ "result.prop1", "result.prop2" ]`	An array of one or more JSONPath strings that reference one or more values, which are returned as an array.
value	any	`"fasttrack"`	A static value that is used as the referenced value. This is an alternative to referencing values with `path` or `paths`.
regex	RegExp	`/[ft].?/`	A regular expression that is applied to the referenced value using the match() method. The first matched value in the match() array is returned. (If no match is found, the referenced value is left unchanged.)
dictionary	Object	`{ fast: "faster", track: "tracker" }`	A dictionary object of key/value pairs. The referenced value is used as a key in the dictionary and the matching value is returned. (If the key does not exist, `undefined` is returned.)
transform	function	`(val) => return val.toUpperCase()`	A function that takes the referenced value and returns a transformed value.
separator	string	`", "`	A separator string that is applied to a referenced array of strings. The array of strings is joined with the separator value.
prefix	string	`"Mr."`	A string prefix prepended to the referenced value.
suffix	string	`", Jr."`	A string suffix appended to the referenced value.

PathConfig Property Name	Type	Example	Description
nilValue	any	“Not found”	Value to use for referenced values that are undefined or null (arrays are ignored).

If multiple transformation properties are provided in `PathConfig`, the transformations are executed in the following order:

1. regex
2. dictionary
3. transform
4. separator
5. prefix
6. suffix

Examples

The examples show how to reference and then transform values in data using `PathConfig`.

Apply a regular expression

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    path: "result.prop3[0]",
    regex: /[fqz].?/
  }}
/>
```

This above configuration returns “fa”.

Apply a dictionary

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    path: "result.prop3[1]",
    dictionary: {{ fast: "quick", track: "trick" }}
  }}
/>
```

The above configuration returns “trick”.

Apply a transform function

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    path: "result.prop2",
    transform: (val) => return val.toUpperCase()
  }}
/>
```

The above configuration returns “LOGIC”.

Join an array with a separator

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    path: "result.prop3",
    separator: " + "
  }}
/>
```

The above configuration returns “fast + track”.

Add a prefix

```
MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    path: "result.prop1",
    prefix: "Mr. "
  }}
/>
```

The above configuration returns “Mr. Mark”.

Add a suffix

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    path: "result.prop1",
```

```
    suffix: ", Jr."
  }}
/>
```

The above configuration returns “Mark, Jr.”.

Apply multiple transformations

```
<MyWidget
  data={{ result: { prop1: "ml", prop2: "ft" }}}
  config={{
    path: "result.prop2",
    regex: /[fqz].?/,
    dictionary: {{ ml: "Mark Logic", ft: "Fast Track" }},
    transform: (val) => return val.replace(' ', ''),
    prefix: "Hello, ",
    suffix: "!"
  }}
/>
```

The above configuration returns “Hello, FastTrack!”.

Get multiple values as an array

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    paths: [ "result.prop2", "result.prop3[1]" ]
  }}
/>
```

The above configuration returns [“Logic”, “fast”].

Use a value

```
<MyWidget
  data={{ result: { prop1: "Mark", prop2: "Logic", prop3: ["fast", "track"]
}}}
  config={{
    value: "default"
  }}
/>
```

The above configuration returns “default”. Nothing in the data object is referenced.

5

Include document extracts in search results

By default, `/v1/search` results do not include data from matching documents. Include this information by using the `extract-document-data` property in the query options of the application.

Matching document

Consider a matching document with this structure:

```
{ "envelope": {
    "entityType": "person",
    "id": 1001,
    "firstName": "Nerta",
    "lastName": "Hallwood",
    "title": "Marketing Manager"
  }
}
```

JSON query options

This example uses JSON query options and includes data from the `/envelope` path in the document for each result. The `extract-path` property takes an XPath expression that selects the document content to

include in the results:

```
{
  "options": {
    "extract-document-data": {
      "selected": "include",
      "extract-path": "/envelope"
    }
  }
}
```

XML query options

XML can also be used to include document content using query options:

```
<?xml version="1.0" encoding="UTF-8"?>
<options>
  <extract-document-data selected="include">
    <extract-path>/envelope</extract-path>
  </extract-document-data>
</options>
```

Search results

When query options are included, these search results are returned:

> **Note:**
>
> Notice that the `extracted` property contains matching document information.

```
      {
          "snippet-format": "snippet",
          "total": 1,
          "start": 1,
          "page-length": 10,
          "selected": "include",
          "results": [
              {
                  "index": 1,
                  "uri": "/person/1001.json",
                  "path": "fn:doc(\"/person/1001.json\")",
                  "score": 2816,
```

```
                    "confidence": 0.2636895,
                    "fitness": 0.2763854,
                    "href": "/v1/documents?uri=%2Fperson%2F1001.json",
                    "mimetype": "application/json",
                    "format": "json",
                    "matches": [
                        {
                            "path": "fn:doc(\"/person/1001.json\")/envelope/
text(\"entityType\")",
                            "match-text": [
                                {
                                    "highlight": "person"
                                }
                            ]
                        }
                    ],
                    "extracted": {
                        "kind": "array",
                        "content": [
                            {
                                "envelope": {
                                    "entityType": "person",
                                    "id": 1001,
                                    "firstName": "Nerta",
                                    "lastName": "Hallwood",
                                    "title": "Marketing Manager",
                                }
                            }
                        ]
                    }
                }
            ],
            "qtext": "person",
            "metrics": {
                "query-resolution-time": "PT0.001205S",
                "snippet-resolution-time": "PT0.001033S",
                "extract-resolution-time": "PT0.000818S",
                "total-time": "PT0.22377S"
            }
        }
```

When document extract information is included in search results, the widgets displaying the results have access to document information. See extract-document-data for more details.

Note:

To include all of the matching documents in search results, include an `extract-document-data` property or element without the `extract-path` property or element. This can be useful

when a JSON document does not have a single `root` property.

6

Add a search result popup window

FastTrack widgets can display the details of a search result in a popup window. Widgets can recognize when a search result is clicked, retrieve the document associated with a clicked result, and display the document content using the WindowCard and EntityRecord widgets.

Example document

> **Note:**
>
> These steps assume you have set up a FastTrack-enabled search application using the steps in Create a search application.

This is an example document from Create a search application:

```
{ "envelope": {
    "entityType": "person",
    "id": 1001,
    "firstName": "Nerta",
```

```
    "lastName": "Hallwood",
    "title": "Marketing Manager",
    "status": "active",
    "dob": "1985-03-04",
    "salary": 104000,
    "address": {
      "street": "40 Summer Ridge Point",
      "city": "Cincinnati",
      "state": "Ohio",
      "country": "United States",
      "latitude": 39.1848,
      "longitude": -84.3448
    },
    "image": "person-1001.jpg",
    "content": "And this, our life, exempt from public haunt, finds tongues
in trees, books in the running brooks, sermons in stones, and good in
everything.",
    "relations": [
      {
        "triple": {
          "subject": "http://example.org/1001",
          "predicate": "http://xmlns.com/foaf/0.1/knows/",
          "object": {
              "datatype" : "http://www.w3.org/2001/XMLSchema#string",
              "value":"1002"
            }
          }
      },
      {
        "triple": {
          "subject": "http://example.org/1001",
          "predicate": "http://xmlns.com/foaf/0.1/knows/",
          "object": {
              "datatype" : "http://www.w3.org/2001/XMLSchema#string",
              "value":"1003"
            }
          }
      }
    ]
  }
}
```

Handle clicks and display results

In a FastTrack-enabled search application, when a search result is clicked, the properties from the search result document can be displayed. To do this, In the search application, open the `App.jsx` file and replace the existing code with this code:

```
import React from 'react'
import './App.css'
import { useContext, useState } from "react";
import { MarkLogicContext, SearchBox, ResultsSnippet, WindowCard,
EntityRecord } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);
  const [showWindow, setShowWindow] = useState(false);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const handleResultClick = (snippet) => {
    context.getDocument(snippet.uri).then((response) => {
      setShowWindow(true);
    })
  }

  const handleWindowClose = () => {
    context.setDocumentResponse(null);
    showWindow && setShowWindow(null);
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div>
        <ResultsSnippet
          results={context.searchResponse.results}
          onClick={handleResultClick}
        />
      </div>
      <div>
        <WindowCard
          title="Person Details"
          visible={showWindow}
          toggleDialog={handleWindowClose}
          initialLeft={640}
          height={240}
          width={320}
        >
          <EntityRecord
            entity={context.documentResponse}
            config={{
              entityTypeConfig: { "path": "data.envelope.entityType" },
              entities: [
                {
```

```
                  entityType: 'person',
                  items: [
                    { label: 'First Name', path: 'data.envelope.firstName' },
                    { label: 'Last Name', path: 'data.envelope.lastName' },
                    { label: 'Title', path: 'data.envelope.title' },
                    { label: 'DOB', path: 'data.envelope.dob' }
                  ]
                }
              ]
            }}
          />
        </WindowCard>
      </div>
    </div>
  )
}

export default App
```

Code explanation

The code in Handle clicks and display results performs these functions:

Adds the WindowCard widget to display the search result details

- The visibility of the window is handled by the state variable `showWindow`. The variable is initially set to `false`.
- The callback prop `toggleDialog` handles closing the window. Other props set the window's title, dimensions, and placement in the browser.

Adds the EntityRecord widget inside the WindowCard widget to display the document information

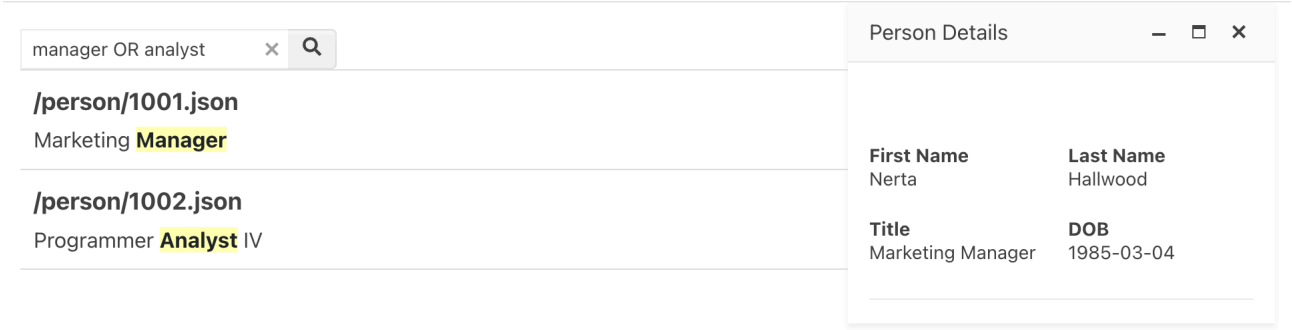
- The `entity` prop sets the retrieved document data (which is stored in the context after retrieval).
- The `config` prop specifies the document properties to display.
- See the EntityRecord documentation for more information about configuration.

Adds an `onClick` callback prop to the ResultsSnippet widget to handle search result clicks

- The clicked search result is passed to the callback function.
- The callback function retrieves the corresponding document using the `getDocument()` method from the context and the document URI from the search result.
- After the document is successfully retrieved, the callback sets `showWindow` to `true` to open the window.

Rendered result

When a search result is clicked, a window opens and displays the search result content.



7

Add faceted search to an application

The FastTrack StringFacet widget is used to add faceted search to an application. Faceted search allows users to narrow search results by placing constraints on properties in the results documents.

Existing search application

These steps in this section assume a FastTrack-enabled search application has been set up using the steps in Create a search application.

Example document

This is an example document from the Create a search application section:

```
{ "envelope": {
    "entityType": "person",
    "id": 1001,
    "firstName": "Nerta",
    "lastName": "Hallwood",
```

```
    "title": "Marketing Manager",
    "status": "active",
    "dob": "1985-03-04",
    "salary": 104000,
    "address": {
      "street": "40 Summer Ridge Point",
      "city": "Cincinnati",
      "state": "Ohio",
      "country": "United States",
      "latitude": 39.1848,
      "longitude": -84.3448
    },
    "image": "person-1001.jpg",
    "content": "And this, our life, exempt from public haunt, finds tongues
in trees, books in the running brooks, sermons in stones, and good in
everything.",
    "relations": [
      {
        "triple": {
          "subject": "http://example.org/1001",
          "predicate": "http://xmlns.com/foaf/0.1/knows/",
          "object": {
              "datatype" : "http://www.w3.org/2001/XMLSchema#string",
              "value":"1002"
            }
          }
      },
      {
        "triple": {
          "subject": "http://example.org/1001",
          "predicate": "http://xmlns.com/foaf/0.1/knows/",
          "object": {
              "datatype" : "http://www.w3.org/2001/XMLSchema#string",
              "value":"1003"
            }
          }
      }
    ]
  }
}
```

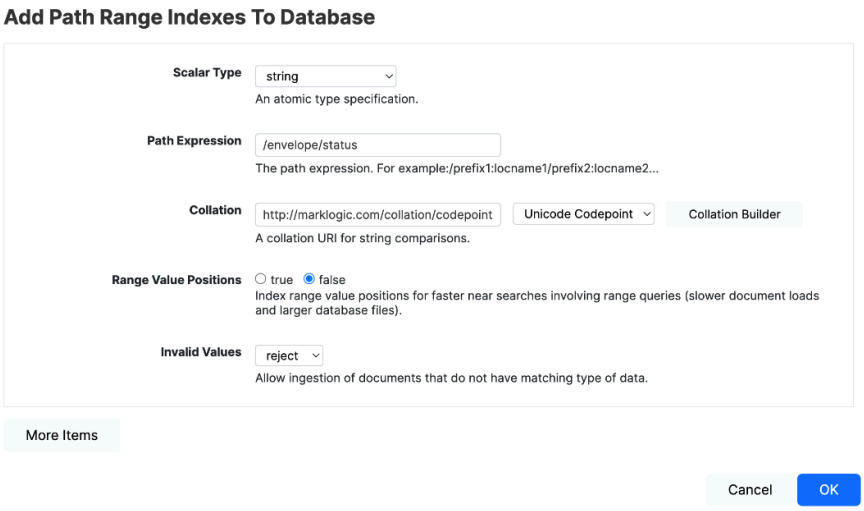
MarkLogic setup

Faceted search in MarkLogic requires a range index on the faceted property.

To add a path range index to the database configuration using the Admin Interface:

1. With MarkLogic running, access the Admin Interface on port 8081. For example, http://localhost:8001.

2. On the left-side of the screen, click **Databases**.
3. Click the **Documents** database.
4. On the left-side of the screen, under **Documents**, click **Path Range Indexes**.
5. Click the **Add** tab.
6. Configure a path range index for the data in the database.



7. Click **OK** to save the index.

Configure a constraint in the query options

After configuring a range index, a constraint in the query options can be configured so that the search application returns facet information for the property in the search results. The query options for the search application can be edited in QConsole. QConsole can be accessed on port 8000 when MarkLogic is running (for example: http://localhost:8000).

To configure a constraint in the query options:

1. In QConsole, select **Modules** from the **Databases** menu, and then click **Explore**.
2. Click the document with the URI /**Default/App-Services/rest-api/options/search-options.xml** and then click **Edit**.
3. Replace the XML content for the query options. This XML adds a facet constraint corresponding to the index defined previously.

```
<?xml version="1.0" encoding="UTF-8"?>
<options xmlns="http://marklogic.com/appservices/search">
<transform-results apply="snippet">
```

```
    <per-match-tokens>30</per-match-tokens>
    <max-matches>4</max-matches>
    <max-snippet-chars>200</max-snippet-chars>
</transform-results>
<constraint name="status">
    <range type="xs:string" facet="true" collation="http://marklogic.com/
collation/codepoint">
    <path-index>/envelope/status</path-index>
    <facet-option>frequency-order</facet-option>
    <facet-option>descending</facet-option>
    </range>
</constraint>
<return-facets>true</return-facets>
</options>
```

4. Click **Save** to save the updated query options.

5. With the query options updated, MarkLogic will return facet information for the status constraint in the search response. For example:

```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "score": 0,
            "confidence": 0,
            "fitness": 0,
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "mimetype": "application/json",
            "format": "json",
            "matches": [
                {
                    "path": "fn:doc(\"/person/1001.json\")/object-node()",
                    "match-text": [
                        "person Nerta Hallwood Marketing Manager active
1985-03-04 person-1001.jpg"
                    ]
                }
            ],
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
```

```
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
                            "status": "active",
                            "dob": "1985-03-04",
                            "image": "person-1001.jpg",
                        }
                    }
                ]
            }
        },
        // ...
    ],
    "facets": {
        "status": {
            "type": "xs:string",
            "facetValues": [
                {
                    "name": "inactive",
                    "count": 2,
                    "value": "Inactive"
                },
                {
                    "name": "active",
                    "count": 1,
                    "value": "Active"
                }
            ]
        }
    },
    // ...
}
```

Add the StringFacet widget

After configuring a constraint in the query options, add the StringFacet widget.

To add the string facet widget:

1. In the search application, open the `App.jsx` file.
2. Replace the existing content with this code:

```
import React from 'react'
import './App.css'
```

```
import { useContext } from "react";
import { MarkLogicContext, SearchBox, ResultsSnippet, StringFacet } from
"ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const handleFacetClick = (selection) => {
    context.addStringFacetConstraint(selection)
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div style={{display: 'flex', flexDirection: 'row'}}>
          <div style={{width: '640px'}}>
            <ResultsSnippet
              results={context.searchResponse.results}
              paginationFooter={true}
            />
          </div>
          <div>
            <StringFacet
              title="Status"
              name="status"
              data={context.searchResponse?.facets?.status}
              onSelect={handleFacetClick}
            />
          </div>
      </div>
    </div>
  )
}

export default App
```

Code explanation

The code in Add the StringFacet widget:

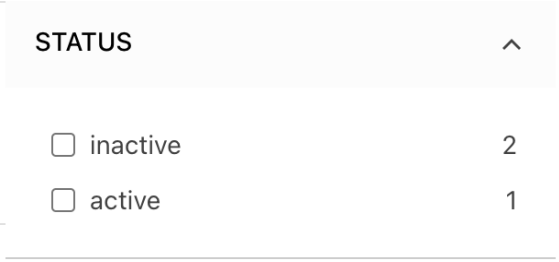
- Imports the StringFacet widget into the application.
- Renders the StringFacet widget with these props:

- The `title` prop gives the facet container the title "Status".
- The `name` prop associates the widget with the constraint named "status".
- The `data` prop defines the facet results in the search response using a JSONPath expression.
- The `onSelect` prop defines the callback function for handling checkbox clicks.

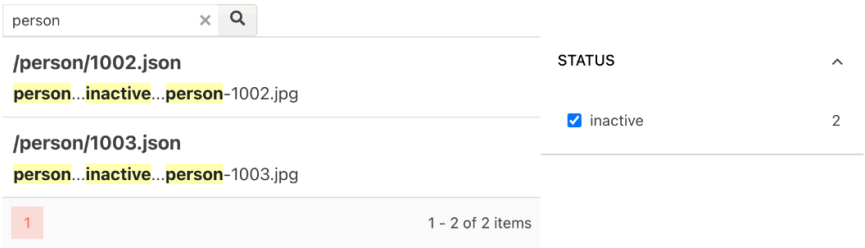
- The `handleFacetClick` callback receives a selection object as an argument. The selection object is then passed into the `addStringFacetConstraint` context method. This adds a facet constraint for the `status` property to the application context. The constraint is then applied to any search request.

Rendered result

This example shows the StringFacet widget. Two returned documents are in inactive status, and one is in active status:



To narrow the search, the user can click one of the checkboxes. In this example, the **inactive** checkbox is selected. As a result, the application only shows inactive documents. The user can uncheck the box to remove the facet constraint.



8

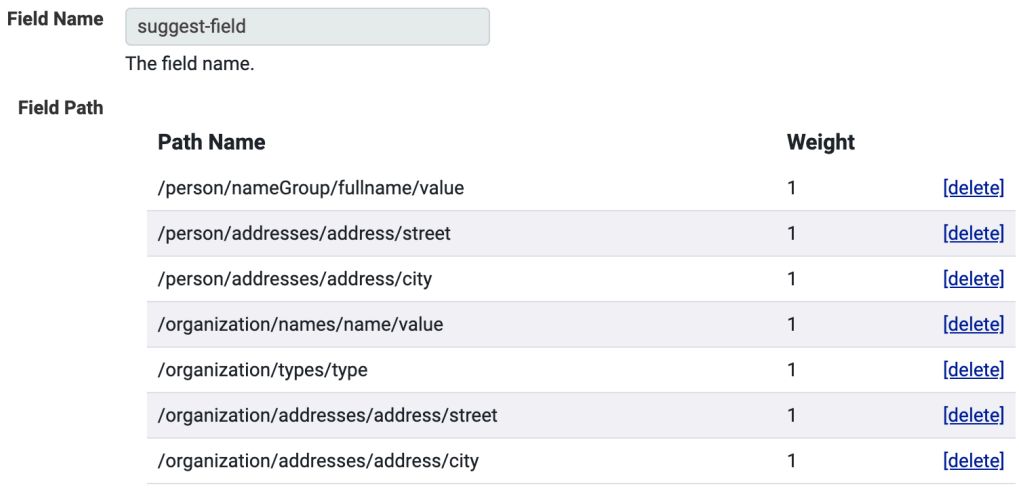
Add typehead search for SearchBox

You can turn on the typeahead search feature in the SearchBox widget to display search suggestions as the user types. By configuring the MarkLogic database and installing query options for suggestions, you can specify what parts of the documents are considered when retrieving suggestions.

Database Configuration

Configure the MarkLogic database to support suggestions. In the example that follows, a field has been defined so that the suggestions come from specific elements in the documents. For more about fields, see Fields Database Settings.

The following is a screenshot from the Admin UI application showing the defined `suggest-field` field:



Field Name: suggest-field

The field name.

Field Path

Path Name	Weight	
/person/nameGroup/fullname/value	1	[delete]
/person/addresses/address/street	1	[delete]
/person/addresses/address/city	1	[delete]
/organization/names/name/value	1	[delete]
/organization/types/type	1	[delete]
/organization/addresses/address/street	1	[delete]
/organization/addresses/address/city	1	[delete]

Query Options Configuration

In your search application's query options, define your suggestion source based on the database configuration, for example:

```
 <options xmlns="http://marklogic.com/appservices/search">
    <default-suggestion-source>
      <range type="xs:string">
        <field name="suggest-field" collation="http://marklogic.com/
collation/"/>
      </range>
    </default-suggestion-source>
  </options>
```

For details, see Install Query Options.

Example SearchBox Configuration

Turn on and configure search suggestions in your SearchBox widget, for example:

```
import { SearchBox } from 'ml-fasttrack'
// ...
<SearchBox
  menuThemeColor="dark"
  buttonThemeColor="light"
  placeholder="Search..."
  searchSuggest={true}
  searchSuggestMin={3}
  searchSuggestSubmit={true}
  searchSuggestLimit={10}
  showLoading={true}
  onSearch={handleSearch}
```

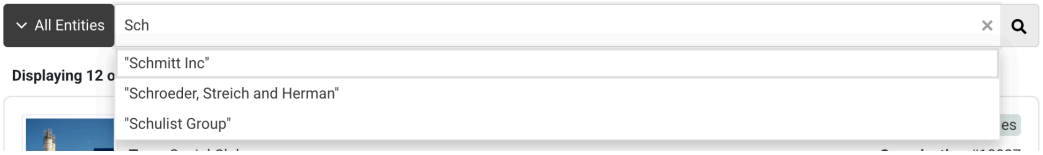
```
  menuItems={searchBox.items}
  value={''}
  containerStyle={{ width: '470px' }}
  boxStyle={{ height: 40 }}
  dropdownItemStyle={{ fontSize: 14 }}
/>
```

For details about SearchBox props, see SearchBox.

Rendered typehead search

When a user types a minimum number of characters in the search box, the search box displays a menu of search suggestions, if any are found. Behind the scenes, MarkLogicContext makes calls to the GET /v1/suggest REST endpoint to retrieve suggestions based on the query options.

With the above code and configuration in place, typing “Sch” in the search box displays a dropdown list of field elements that begin with those characters.



9

FastTrack scenarios

This section contains information on the High-value Target and Crime Map scenarios.

Scenario 1: High-value Target

The High-value Target example application lets you explore connections between potential terrorists. It is based on the SYNCOIN (synthetic counter insurgency) dataset which includes realistic observation data from Baghdad, Iraq in 2010. Data entities, including people, locations, and events, are displayed on a timeline and network graph.

The High-value Target application is included with FastTrack as `hvt.zip`. A README file describes how to run the application.

Scenario 2: Crime Map

The Crime Map example application lets you search a dataset of fictitious crime reports in San Francisco. You can also examine how the crime reports might relate to other crimes under investigation. You can filter events by keyword, distance from eyewitness locations, and time range. Events are displayed on a geospatial map and timeline.

The Crime Map application is included with FastTrack as `crime-map.zip`. A README file describes how to

run the application.

10

FastTrack widgets

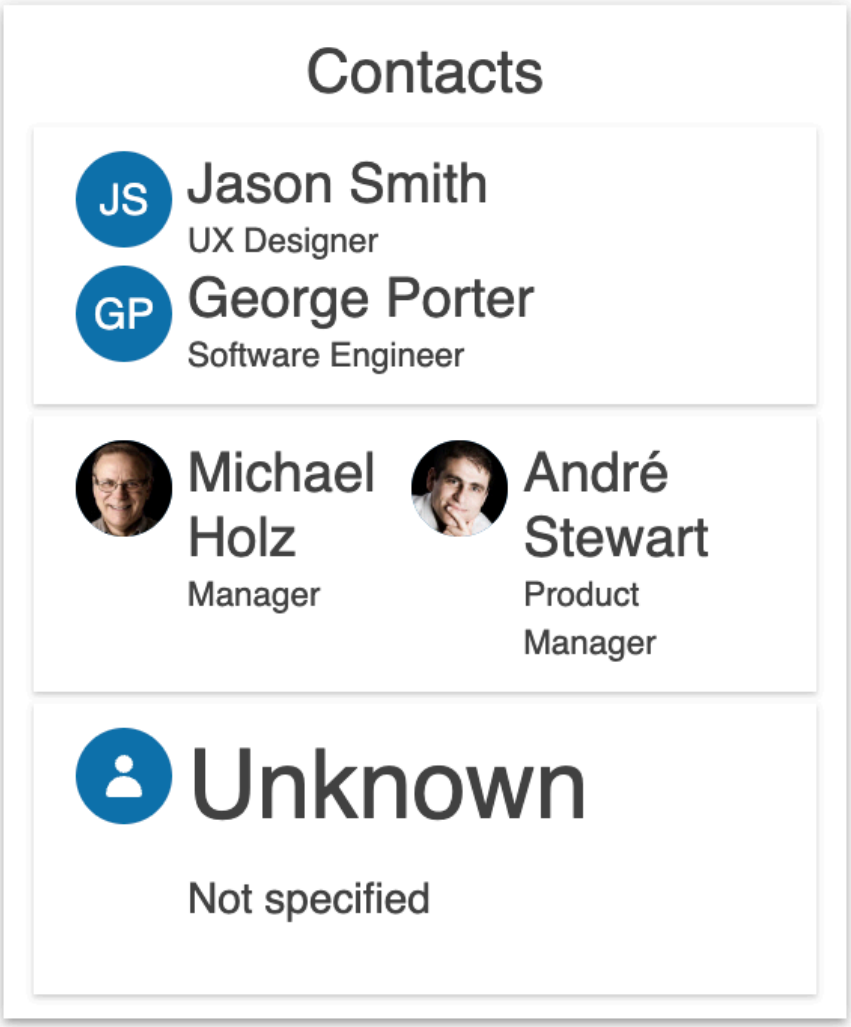
The FastTrack widgets are described in this section.

Avatar

The Avatar widget displays formatted images, icons, or initials that represent people or other entities in an application. The widget is based on the KendoReact Avatar component.

Avatar example rendering

This example shows the Avatar widget with formatted initials, two images, and an icon.



Avatar example configuration

```
import { useContext } from "react";
import './App.css';
import { Avatar } from "ml-fasttrack";

function App() {

    return (
      <div className="App">
        <Avatar
            avatarImage={{ path: 'img', alt: 'Avatar pic' }}
            rounded="full"
            type="image"
            size="medium"
            data={{ img: 'https://www.example.org/myImage.jpg'}}
```

```
          />
        </div>
      );

}

export default App;
```

Avatar code explanation

In the Avatar example configuration the Avatar widget displays an image from a web host.

- The `avatarImage` prop is used with the `data` prop as a configuration object.
- A path points to the image details in the `data` object.
- Alternatively, an image can be displayed using the `children` props to pass in an image component.
- For additional examples, see the KendoReact Avatar documentation.

Avatar API

Prop	Type	Description
border	boolean	Turns the Avatar border on or off. See the KendoReact AvatarProps.
children	ReactNode	Content wrapped by the Avatar widget. See the KendoReact AvatarProps.
className	string	Class name applied to the widget. See the KendoReact AvatarProps.
fillMode	"null" | "solid" | "outline"	Avatar fill mode. See the KendoReact AvatarProps.
rounded	"null" | "small" | "medium" | "large" | "full"	Avatar roundness. See the KendoReact AvatarProps.

Prop	Type	Description
size	"null" | "small" | "medium" | "large"	Avatar size. See the KendoReact AvatarProps.
style	CSSProperties	CSS styles applied to the widget. See the KendoReact AvatarProps.
themeColor	"null" | "base" | "primary" | "secondary" | "tertiary" | "info" | "success" | "warning" | "error" | "dark" | "light" | "inverse"	Avatar theme color. See the KendoReact AvatarProps.
type	"image" | "text" | "icon"	Avatar type. See the KendoReact AvatarProps.
avatarImage	{ path: string; alt?: string; }	Configuration object used when an image is displayed.
avatarImage.alt	string	Alternative text for the image.
avatarImage.path	string	The JSONPath to the image URL in the `data` prop object. Example: `person.images.image[0].url`.
data	object	Data object used with `avatarImage` property. Includes the image URL as a value.

BucketRangeFacet

The BucketRangeFacet widget displays bucketed ranges for a faceted numeric property in search results. Once search results are returned, users can check a box next to the buckets to constrain the results.

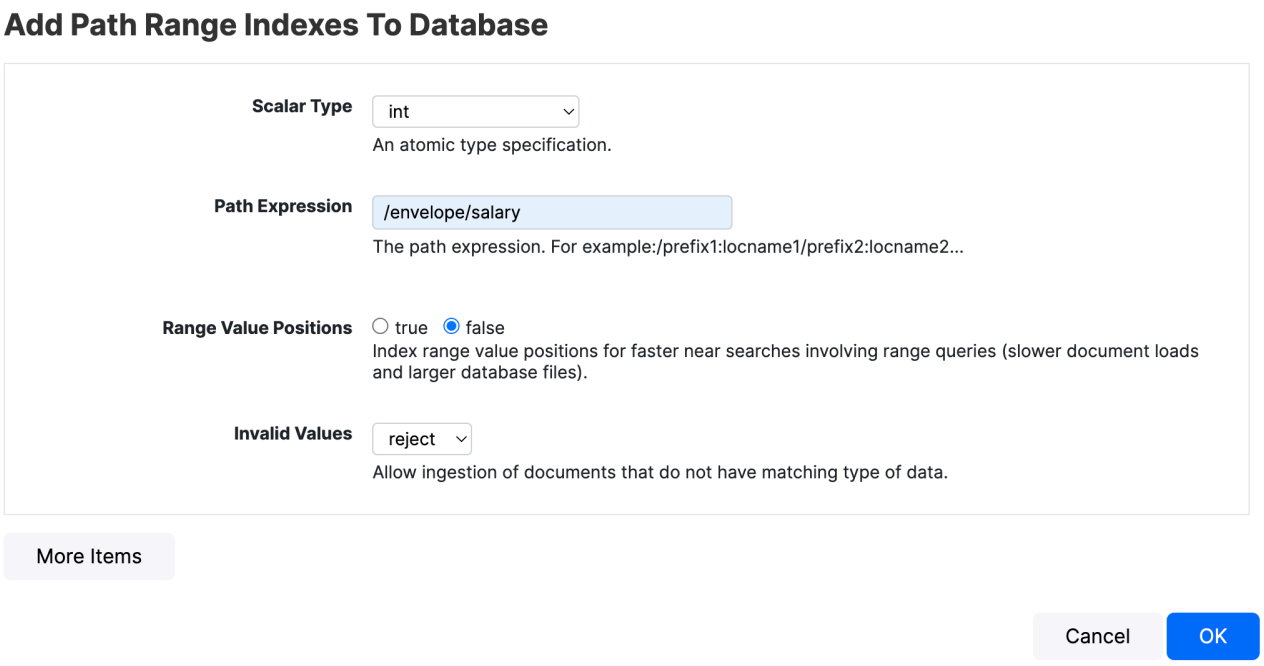
Note:

The NumberRangeFacet widget can also be used to constrain search results for a faceted numeric property.

MarkLogic setup

Faceted search requires a range index on the faceted property. A range index can be added to the database configuration with the MarkLogic Admin Interface or with an API.

This example shows a path range index added to the salary property in the Admin Interface.



After an index is added, a faceted constraint must be configured using query options. In this example, the search application returns facets for the `/envelope/salary` property in the search results. The `constraint` settings correspond to the settings for the index. `return-facets` is set to `true` so that facet results are returned:

```
<?xml version="1.0" encoding="UTF-8"?>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="salaryBucketed">
    <range collation="" facet="true" type="xs:int">
      <path-index>/envelope/salary</path-index>
      <bucket lt="75000" ge="50000" name="$50000 - $75000">$50000 -
$75000</bucket>
      <bucket lt="100000" ge="75000" name="$75000 - $100000">$75000 -
$100000</bucket>
```

```
      <bucket ge="100000" name="Over $100000">Over $100000</bucket>
      <facet-option>limit=25</facet-option>
    </range>
  </constraint>
  <return-facets>true</return-facets>
</options>
```

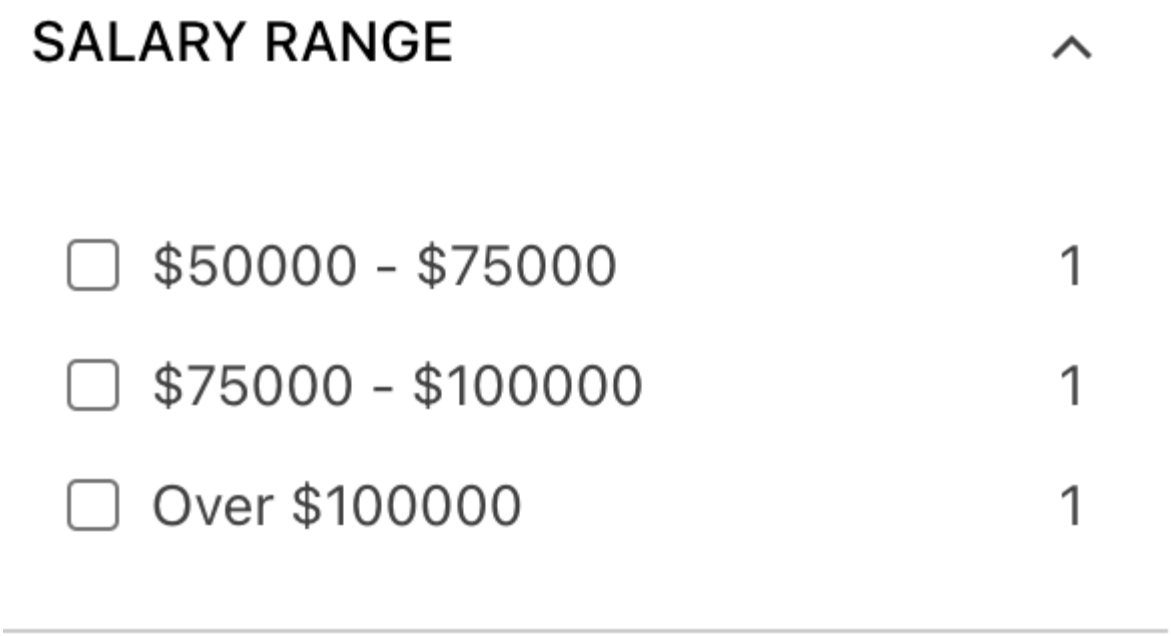
The example code will return this response:

```
{
  "snippet-format": "snippet",
  "total": 3,
  "start": 1,
  "page-length": 10,
  "selected": "include",
  "results": [
    {
      "index": 1,
      "uri": "/person/1001.json",
      "path": "fn:doc(\"/person/1001.json\")",
      "score": 0,
      "confidence": 0,
      "fitness": 0,
      "href": "/v1/documents?uri=%2Fperson%2F1001.json",
      "mimetype": "application/json",
      "format": "json",
      "matches": [
        {
          "path": "fn:doc(\"/person/1001.json\")/object-node()",
          "match-text": [
            "person Nerta Hallwood Marketing Manager Active 1985-03-04 104000
person-1001.jpg"
          ]
        }
      ],
      "extracted": {
        "kind": "array",
        "content": [
          {
            "envelope": {
              "entityType": "person",
              "id": 1001,
              "firstName": "Nerta",
              "lastName": "Hallwood",
              "title": "Marketing Manager",
              "status": "Active",
              "dob": "1985-03-04",
              "salary": 104000,
              "image": "person-1001.jpg",
            }
          }
```

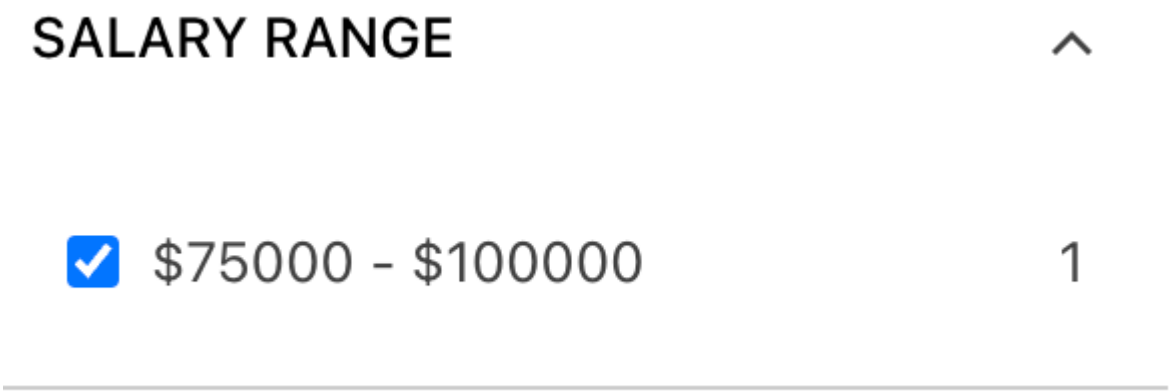
```
        ]
      }
    },
    // ...
  ],
  "facets": {
    "salaryBucketed": {
      "type": "bucketed",
      "facetValues": [
        {
          "name": "$50000 - $75000",
          "count": 1,
          "value": "$50000 - $75000"
        },
        {
          "name": "$75000 - $100000",
          "count": 1,
          "value": "$75000 - $100000"
        },
        {
          "name": "Over $100000",
          "count": 1,
          "value": "Over $100000"
        }
      ]
    }
  },
  // ...
}
```

Example rendering

Using the code and response in MarkLogic setup, the BucketRangeFacet widget displays a set of range buckets based on the results. Each bucket contains one document in the salary range. A user can check the corresponding box to apply the constraint on the facet.



This shows the BucketRangeFacet widget after a user clicks the $75,000–$100,000 range bucket.



BucketRangeFacet example configuration

In this example configuration, the BucketRangeFacet widget is imported and configured in a React application.

```
import { useContext, useState } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsSnippet, BucketRangeFacet } from
"ml-fasttrack";
```

```
function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const handleFacetClick = (selection) => {
    context.addStringFacetConstraint(selection)
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div style={{display: 'flex', flexDirection: 'row'}}>
          <div style={{width: '640px'}}>
            <ResultsSnippet
              results={context.searchResponse.results}
              paginationFooter={true}
            />
          </div>
          <div>
          {context?.searchResponse?.facets?.salaryBucketed &&
            <BucketRangeFacet
              title="Salary Range"
              data={context?.searchResponse?.facets?.salaryBucketed}
              name="salaryBucketed"
              onSelect={handleFacetClick}
            />
          }
          </div>
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In the BucketRangeFacet example configuration:

- The `data` prop is set to the bucketed numeric facet from the search response object.
- The `onSelect` callback manages check box clicks and receives a selection object from the widget.
- The application can then set the facet constraint in the application context using the `addStringFacetConstraint` method.

BucketRangeFacet API

Prop	Type	Description
data	{ type: string; facetValues: { name: string; count: number; value: string; }[]; }	Facet data to display from search results.
name	string	String identifying the facet. Passed as the name value in the onSelect event.
title	string	Title for the collapsible header.
subTitle	string	Subtitle for the collapsible header.
containerStylename	CSSPropertiesstring	String value identifying the item to uncheck. For integration with the SelectedFacets widget.
reset	string	String value identifying the item to uncheck. For integration with the SelectedFacets widget. CSS style for the container
onSelect	(value: { type: string; name: string; value: string[]; title?: string | undefined; }) => voidstring	Callback function triggered by a selection. Receives a selection object.Label for the reset button.

BucketRangeFacet callbacks

This example shows a selection object passed to the `onSelect` callback:

```
{
  "type": "string",
  "name": "salaryBucketed",
  "value": [
    "$50000 - $75000"
```

```
  ],
  "title": "Salary Range"
}
```

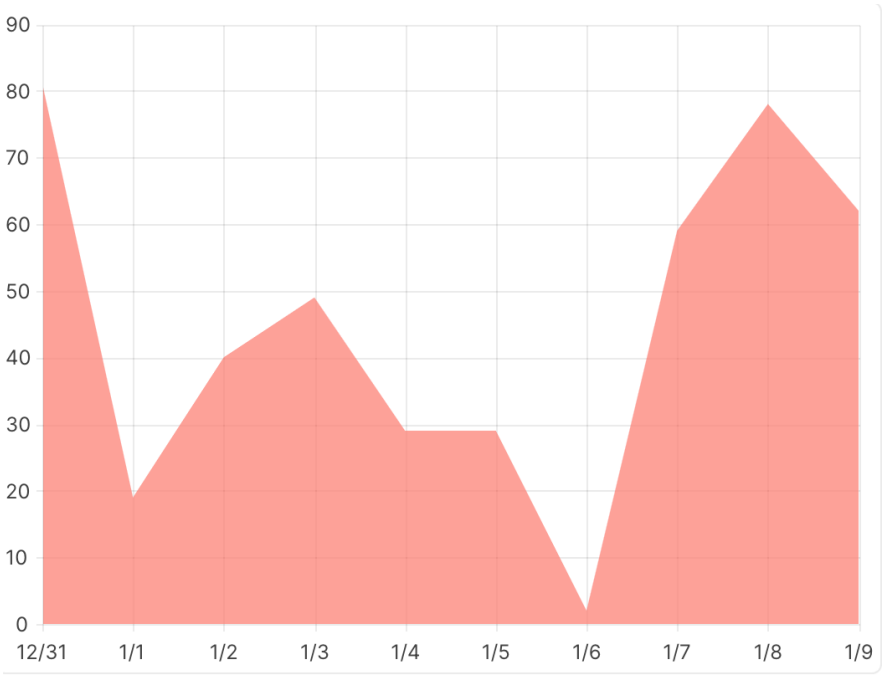
CategoricalChart

The CategoricalChart widget displays charts based on categorical data. Charts are configured by passing settings from the KendoReact Chart component. See API, config API, and *ChartProps API for additional information.

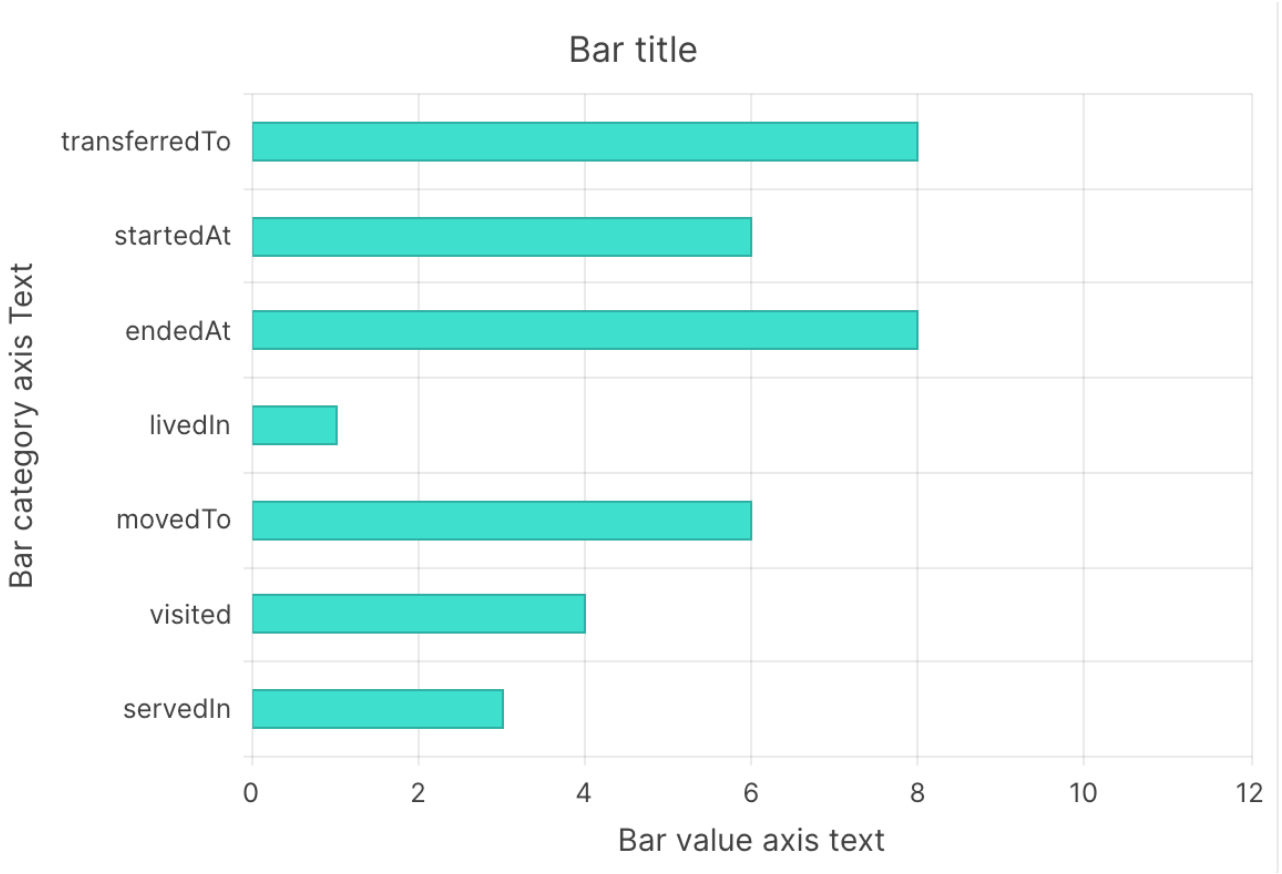
The CategoricalChart widget supports:

- Area charts
- Bar charts
- Column charts
- Donut charts
- Line charts
- Radar line charts
- Pie charts

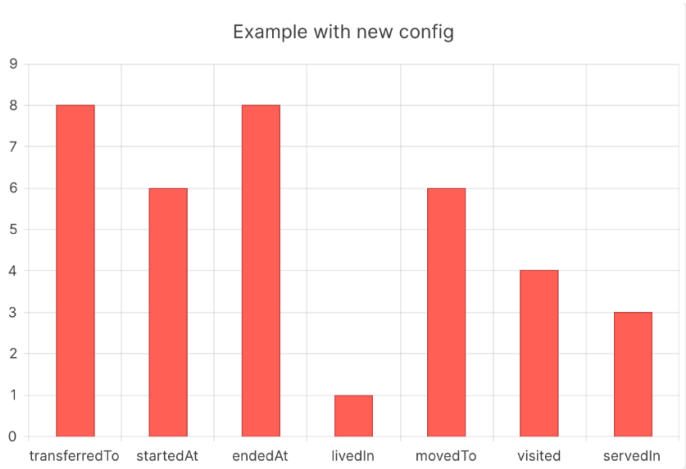
Area chart



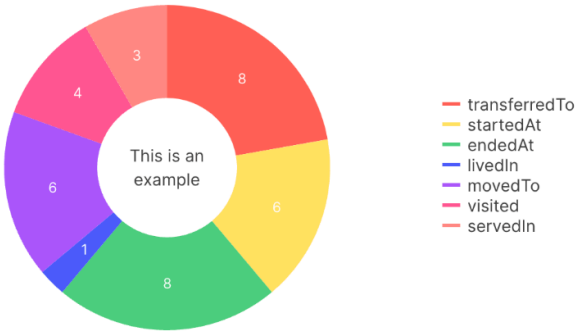
Bar chart



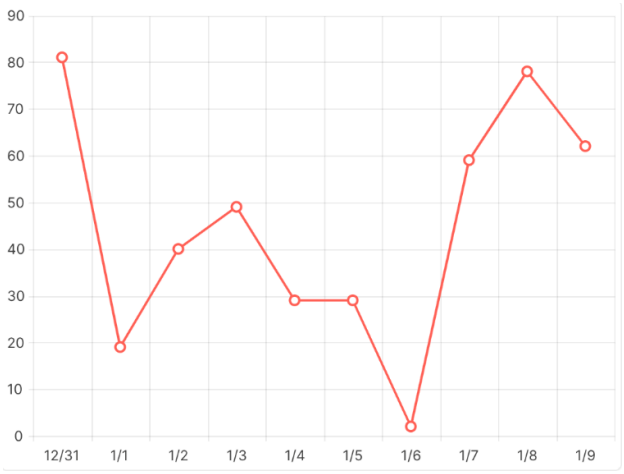
Column chart



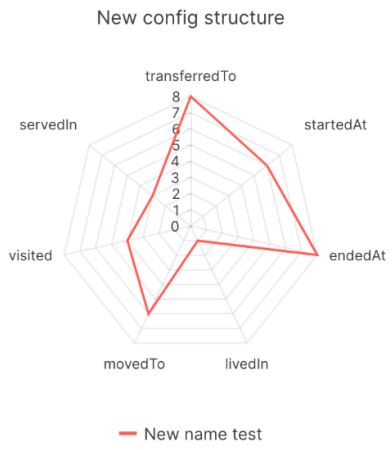
Donut chart



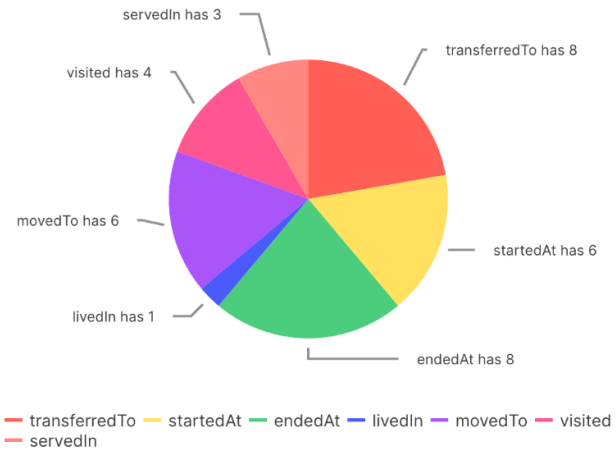
Line chart



Radar line chart



Pie chart



Display formatted data in a chart

The CategoricalChart widget can display formatted data. Formatted data should be passed as an array of objects with value and category properties as shown in this example.

```
import './App.css';
import { CategoricalChart } from "ml-fasttrack";

function App() {

  const dataFormatted = [
    {
```

```
      category: new Date(2024, 0, 1),
      value: 1
    },
    {
      category: new Date(2024, 0, 2),
      value: 2
    }
  ]

  return (
    <div className="App">
      <div style={{width: '480px'}}>
        <CategoricalChart
          data={dataFormatted}
          chartType="line"
        />
      </div>
    </div>
  );
}

export default App;
```

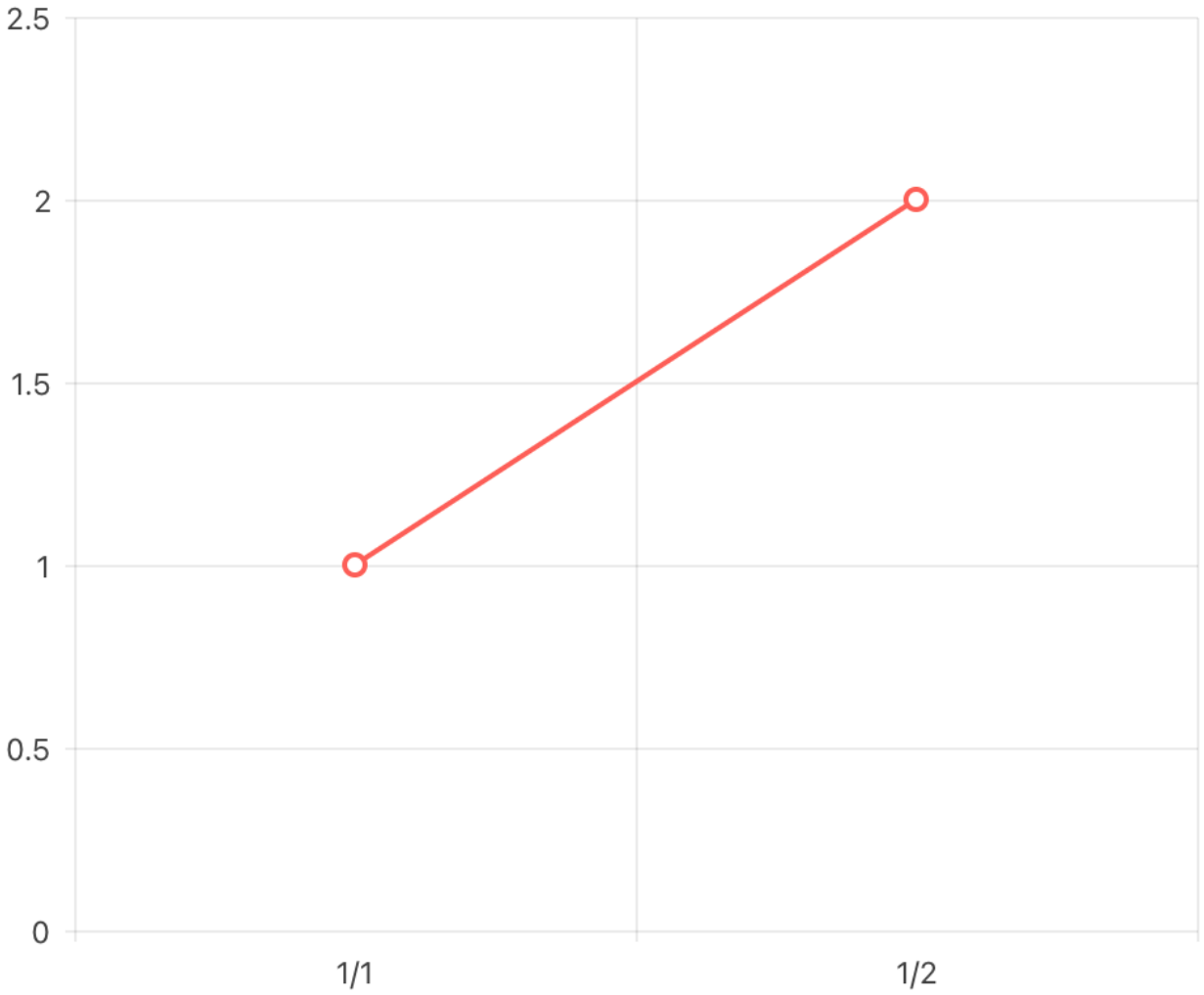
Code explanation

In this example:

- An array of objects is passed as the `data` prop.
- The `chartType` prop specifies the type of chart.

Example rendering

The code in Display formatted data in a chart renders this line chart:



Display transformed data in a chart

Data can be transformed using a transformation function defined with the `transformData` prop. In this example, the code transforms an array of arrays into an array of objects. An array of objects is required by the widget. The `chartType` prop specifies the type of chart:

```
import './App.css';
import { CategoricalChart } from "ml-fasttrack";

function App() {

  const dataRaw = [
    [ "active", 1],
    [ "inactive", 2]
  ]
```

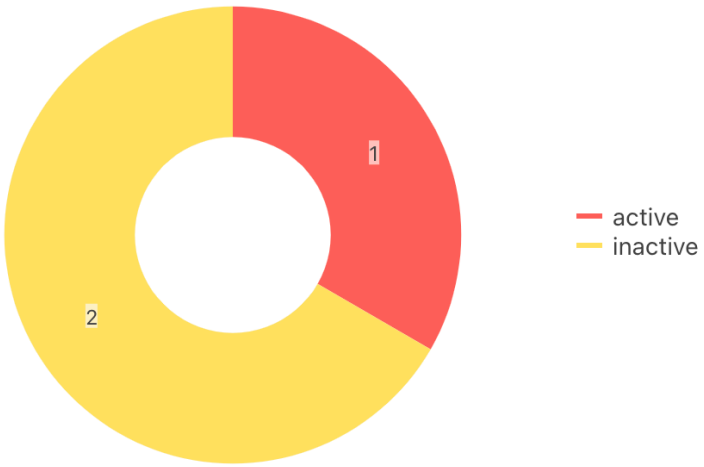
```
  const transformData = (dataArray) => {
    return dataArray.map(item => ({
        category: item[0],
        value: item[1]
    }))
  }

  return (
    <div className="App">
      <div style={{width: '480px'}}>
        <CategoricalChart
          data={dataRaw}
          chartType="donut"
          transformData={transformData}
        />
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In this example:

- The code transforms an array of arrays into an array of objects.
- An array of objects is required by the widget.
- The `chartType` prop specifies the type of chart.

Example rendering



Display search result data in a chart

The CategoricalChart widget can display values from search result documents.

Example search results

This example assumes that an application returns these search results:

> **Note:**
>
> This example also applies to Display facet data in a chart.


```
{
    "total": 3,
    "start": 1,
    "page-length": 10,
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "dob": "1985-03-04",
                            "salary": 104000,
                        }
                    }
                ]
            }
        },
        {
            "index": 2,
            "uri": "/person/1002.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
```

```
                            "id": 1002,
                            "firstName": "Shaylynn",
                            "lastName": "Guard",
                            "dob": "1964-09-30",
                            "salary": 55000
                        }
                    }
                ]
            }
        },
        {
            "index": 3,
            "uri": "/person/1003.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1003,
                            "firstName": "Pris",
                            "lastName": "Sizland",
                            "dob": "1988-12-15",
                            "salary": 87000
                        }
                    }
                ]
            }
        }
    ],
    "facets": {
        "status": {
            "type": "xs:string",
            "facetValues": [
                {
                    "name": "inactive",
                    "count": 2,
                    "value": "inactive"
                },
                {
                    "name": "active",
                    "count": 1,
                    "value": "active"
                }
            ]
        }
    },
    "qtext": "person"
```


These search results have been configured to return document extracts and return facet information for the status property by setting query options. This allows visualizing such data with the widget.

React application code

This is the React application code for rendering a search box and a chart:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, CategoricalChart } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const chartConfig = {
    entityTypeConfig: {
        "path": "extracted.content[0].envelope.entityType"
    },
    entities: [
        {
            entityType: 'person',
            items: [
                {
                    path: 'extracted.content[0].envelope',
                    key: 'status'
                }
            ],
        }
    ]
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
        <div style={{width: '480px'}}>
          <CategoricalChart
            data={context.searchResponse?.results}
            chartType="column"
            config={chartConfig}
            settings={{
              columnChartProps: {
                chartProps: { pannable: true, zoomable: true },
                chartTitleProps: { text: "Statuses" },
                chartSeriesItemProps: { color: "#0cc" }
              }
            }}
            onSeriesClick={(event) => console.log(event)}
          />
```

```
        </div>
    </div>
  );
}

export default App;
```

Code explanation

- The search results to chart are specified with the `data` prop. To do this, use the search response object stored in the application context.
- The `chartType` prop specifies a column chart.
- The `config` prop defines what properties in the payload will appear in the chart for each entity type. For configuration details, see API, config API.
- The `settings` prop defines the KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type. For configuration details, see API.

Example rendering

The React application code displays this chart:

Display facet data in a chart

The CategoricalChart can render a chart that displays facet values from search results. This is similar to displaying facet values using the StringFacet widget but using a chart. The user can click the chart to apply a facet constraint.

Example search results

This example assumes that an application returns these search results:

```
{
    "total": 3,
    "start": 1,
    "page-length": 10,
```

```
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "dob": "1985-03-04",
                            "salary": 104000,
                        }
                    }
                ]
            }
        },
        {
            "index": 2,
            "uri": "/person/1002.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1002,
                            "firstName": "Shaylynn",
                            "lastName": "Guard",
                            "dob": "1964-09-30",
                            "salary": 55000
                        }
                    }
                ]
            }
        },
        {
            "index": 3,
            "uri": "/person/1003.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1003,
                            "firstName": "Pris",
                            "lastName": "Sizland",
```

```
                        "dob": "1988-12-15",
                        "salary": 87000
                    }
                }
            ]
        }
    }
],
"facets": {
    "status": {
        "type": "xs:string",
        "facetValues": [
            {
                "name": "inactive",
                "count": 2,
                "value": "inactive"
            },
            {
                "name": "active",
                "count": 1,
                "value": "active"
            }
        ]
    }
},
"qtext": "person"
```

These search results have been configured to return document extracts and return facet information for the status property by setting query options. This allows visualization of the data with the widget.

React application code

This is the React application code for rendering a search box and a chart:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, CategoricalChart } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const chartFacetConfig = {
    facet: {
        name: "status",
```

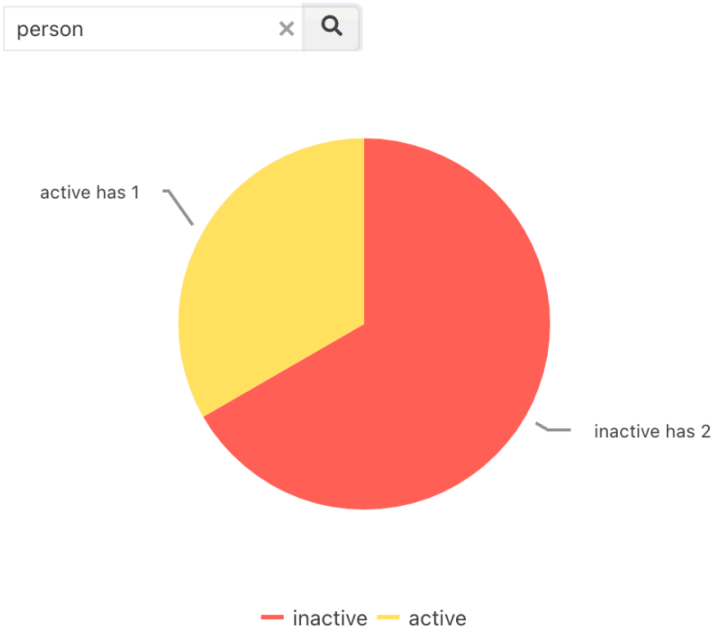
```
        title: "Status"
    }
  }

  const handleChartFacetClick = (event) => {
    context.addStringFacetConstraint({
      type: 'string',
      title: event.target.props.config.facet.title,
      name: event.target.props.config.facet.name,
      value: [event.category]
    })
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
        <div style={{width: '480px'}}>
          <CategoricalChart
            data={context?.searchResponse?.facets}
            chartType="pie"
            config={chartFacetConfig}
            settings={{
              pieChartProps: {
                chartProps: { pannable:true, zoomable: true },
                chartSeriesItemProps: {
                    labels: {
                        visible: true,
                        content:  (props) => {
                            return `${props.dataItem.category} has
${props.dataItem.value}`
                        }
                    }
                },
                chartLegendProps: {
                    position: 'bottom',
                    visible: true
                }
            }}}
            onSeriesClick={handleChartFacetClick}
          />
        </div>
    </div>
  );
}

export default App;
```

Code explanation

- The facet information for the chart is determined by the `config` prop's `facet` property. In this example, the `status` property is charted.
- The `chartType` prop specifies the chart as a pie chart.
- The `onSeriesClick()` event handler is executed when the user clicks a slice in the pie chart. The event handler calls the `addStringFacetConstraint()` method in the application context to set a facet constraint for the status property. For more details, see API.
- The `settings` prop defines the KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type. For configuration details, see API.

Example rendering

The code in this section renders this chart:



When a user clicks a slice in the pie chart, the application context executes a new search with a facet constraint applied. This updates the chart. The SelectedFacets widget can also be added to provide a way of removing facet constraints.

API

Prop	Type	Description
data	object	Data to display in the chart.
config	object	Array of entity configuration objects. Each object determines what data to display from a result for an entity type.
chartType	"area" | "bar" | "column" | "donut" | "line" | "radarLine" | "pie"	The chart type.
settings	object	KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type: • `areaChartProps` • `barChartProps` • `columnChartProps` • `donutChartProps` • `lineChartProps` • `pieChartProps` • `radarLineChartProps` For details, see the *ChartProps API table below and the KendoReact Chart documentation.
onPlotAreaClick	function	Callback function triggered when the chart plot area is clicked. See KendoReact ChartProps.
onSeriesClick	function	Callback function triggered when the

Prop	Type	Description
		chart series is clicked. See KendoReact ChartProps.
style	CSSProperties	CSS styles applied to the chart. See KendoReact ChartProps.
transformData	((data: any) => SeriesData[])	Callback function for transforming the data value. Used only when **not** using a `config` prop. Receives a data object as an argument and returns an array of objects with category and value properties.

config API

Prop	Type	Description
entityTypeConfig	PathConfig	Entity type configuration object.
entityTypeConfig.path	string	Path to the entity type in the search result. Specified using JSONPath.
entities[]	object[]	Array of chart configuration objects for each entity.
entities[].entityType	string	Entity type of the configuration object.
entities[].items[]	PathConfig[]	Array of item configuration objects for the charted data. An array allows for multiple property values in the charted data.
entities[].items[].path	string	Path to the object in the search result with the charted values. Specified using JSONPath.

Prop	Type	Description
entities[].items[].key	string	Property key for the charted value.
facet	object	Facet configuration object.
facet.name	string	Name of the facet values displayed in the chart.
facet.title	string	Title of the facet. This value is required for setting facet constraints in the application context.

ChartProps API

Prop	Type	Description
chartProps	object	ChartProps for the KendoReact Chart component.
chartTitleProps	object	ChartTitleProps for the KendoReact Chart component.
chartSeriesProps	object	ChartSeriesProps for the KendoReact Chart component.
chartSeriesItemProps	object	ChartSeriesItemProps for the KendoReact Chart component.
chartSeriesLabelsProps	object	ChartSeriesLabelsProps for the KendoReact Chart component.
chartValueAxisProps	object	ChartValueAxisProps for the KendoReact Chart component.
chartValueAxisItemProps	object	ChartValueAxisItemProps for the KendoReact Chart component.
chartCategoryAxisProps	object	ChartCategoryAxisProps for the KendoReact Chart component.
chartCategoryAxisItemProps	object	ChartCategoryAxisItemProps for the KendoReact Chart component.
chartLegendProps	object	ChartLegendProps for the KendoReact Chart component.

Prop	Type	Description
chartTooltipProps	object	ChartTooltipProps for the KendoReact Chart component.
chartSeriesItemTooltipProps	object	ChartSeriesItemTooltipProps for the KendoReact chart component.

CommentBox

The CommentBox widget allows users to submit comments about an entity instance. The widget includes an input field and submit button. When the submit button is clicked, an event handler can be used to save comments in MarkLogic. An image can also be included to identify users who submit comments.

The CommentBox widget typically works alongside the CommentList widget. The CommentList widget displays the comments associated with an entity instance.

CommentBox MarkLogic setup

To use the CommentBox widget to store comments in documents, the documents must have the appropriate permissions. See Protecting Documents for details.

Rendering a comment box

This example React application displays a CommentBox widget. The code is configured to save a user's comment in MarkLogic using the application context.

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, CommentBox} from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const onSubmitComment = async (comment) => {
    let res = await context.patchComment(
      '/person/1001.json', { content: comment, context: "/envelope/array-
node('comments')" }
    )
    if (res.success === true) console.log("Comment submitted");
  }

  return (
    <div className="App">
```

```
      <div>
        <div style={{height: '240px'}}>
          <CommentBox
            label="Comments"
            inputPlaceholder="Add a comment"
            buttonLabel="Submit"
            username="a-user"
            imgSrc="https://demos.telerik.com/kendo-ui/content/web/Customers/
RICSU.jpg"
            profileImage={{
              alt: 'Avatar',
              path: 'https://demos.telerik.com/kendo-ui/content/web/
Customers/RICSU.jpg'
            }}
            onSubmit={(comment) => onSubmitComment(comment)}
          />
        </div>
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In the Rendering a comment box code:

- The `username` prop identifies the user who submits a comment. Typically, this value is dynamically set to the currently logged-in user. In this example, the value is hardcoded for simplicity.
- The `onSubmit` prop accepts a callback function that saves the comment to MarkLogic. In this example, the callback executes the `MarkLogicContext.patchComment()` method to update the entity instance document.
- For details about the other props available for CommentBox, see CommentBox API.

Saved comments

When the CommentBox and CommentList widgets are used with the `MarkLogicContext` methods, the comments can be saved in either XML or JSON.

Saved comments (XML)

In XML documents, comments are stored as children of an element in the document:

```
<comments>
  <comment id="cfbb02bc-680f-43f3-b0d3-2215aba5bc07">
```

```
    <content>This is the first comment.</content>
    <ts>2024-01-28T03:30:24.871Z</ts>
    <username>joe-user</username>
    <imgSrc>https://demos.telerik.com/kendo-ui/content/web/Customers/
RICSU.jpg</imgSrc>
  </comment>
  <comment id="4487fb6b-4605-4822-947c-0e8b0964ae4a">
    <content>This is the second comment.</content>
    <ts>2024-01-28T03:41:47.034Z</ts>
    <username>joe-user</username>
    <imgSrc>https://demos.telerik.com/kendo-ui/content/web/Customers/
RICSU.jpg</imgSrc>
  </comment>
</comments>
```

Saved comments (JSON)

In JSON documents, the comments are stored as an array of comment objects:

```
"comments": [
  {
     "id": "cfbb02bc-680f-43f3-b0d3-2215aba5bc07",
     "content": "This is the first comment.",
     "ts": "2024-01-28T03:30:24.871Z",
     "username": "joe-user",
     "imgSrc": "https://demos.telerik.com/kendo-ui/content/web/Customers/
RICSU.jpg"
  },
  {
     "id": "4487fb6b-4605-4822-947c-0e8b0964ae4a",
     "content": "This is the second comment.",
     "ts": "2024-01-28T03:41:47.034Z",
     "username": "joe-user",
     "imgSrc": "https://demos.telerik.com/kendo-ui/content/web/Customers/
RICSU.jpg"
  }
]
```

CommentBox example rendering

The Rendering a comment box code displays this comment box:

CommentBox API

Prop	Type	Properties
label	string	Label appearing above the comment box.
buttonLabel	string	Label for the button for submitting comments.
inputPlaceholder	string	Placeholder text displayed in the comment input field.
onSubmit	(comment: { content: string, ts: string, username: string, imgSrc: string }) => void	Callback function triggered when the user clicks the submit button. It receives an object representing the comment data.
onChange	(comment: any) => void	Callback function triggered when the user changes the comment input field. It receives the current field text as an argument.
username	string	Username of the commenter.
profileImage	{ path: string; alt: string; }	Object for configuring the comment avatar. A `path` property defines the avatar image URL, and an `alt` property defines the alternative text.
imgSrc	string	Optional URL string for the image associated with the commenter. It is submitted as part of the comment payload.
renderImage	ReactNode	Optional custom React component for rendering the avatar image next to

Prop	Type	Properties
		the comment box. If provided, it will override the default image.
renderButton	ReactNode	Optional custom React component for rendering the submit button. If provided, it will override the default button.
containerStyle	CSSProperties	CSS styles applied to the widget.
labelStyle	CSSProperties	CSS styles applied to the comment label.
imageStyle	CSSProperties	CSS styles applied to the avatar image.

CommentList

The CommentList widget displays a list of comments associated with an entity instance. The widget also displays the user name, time stamp, and any image associated with a comment. A sort menu allows users to sort the comments by time stamp. Comments can be edited and deleted by clicking a link. CommentList is often implemented with the CommentBox widget. The CommentBox widget displays both an input box and a submit button.

> **Note:**
>
> Define an event handler to edit and delete comments.

CommentList MarkLogic setup

In order to use the CommentBox and CommentList widgets to manage document comments, the documents must have the appropriate read and update permissions. See Protecting Documents for details.

CommentList example

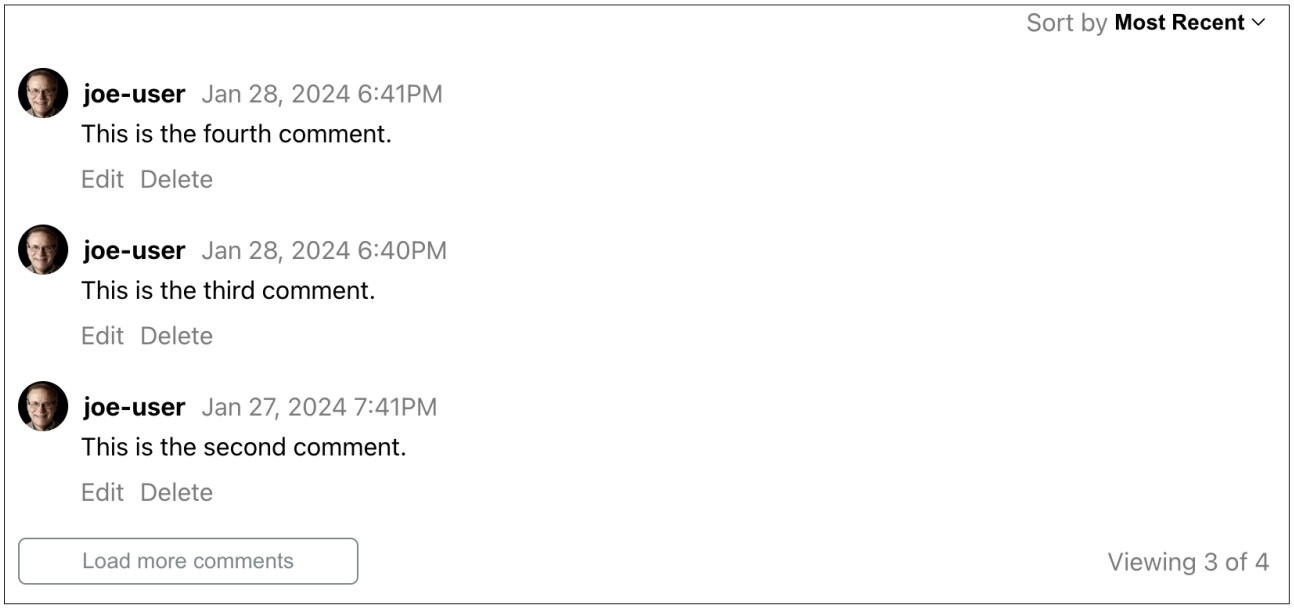
This React application displays a CommentList widget (along with a CommentBox widget for submitting comments):

```
import { useContext, useEffect } from "react";
import './App.css';
import { MarkLogicContext, CommentBox, CommentList } from "ml-fasttrack";

function App() {
  const context = useContext(MarkLogicContext);
  useEffect(() => {
    context.getDocument('/person/1001.json');
  }, []);
  const onSubmitComment = async (comment) => {
    let res = await context.patchComment(
      '/person/1001.json', { content: comment, context: "/envelope/array-
node('comments')" }
    )
      if (res.success === true) context.getDocument('/person/1001.json'); //
Reload after new comment
    }
  const onEditComment = async (commentId, comment) => {
    let res = await context.editComment(
      '/person/1001.json', commentId, { content: comment, context:
"/envelope/array-node('comments')" }
    )
    if (res.success === true) context.getDocument('/person/1001.json'); //
Reload after edit
  }
  const onDeleteComment = async (commentId) => {
    let res = await context.deleteComment(
      '/person/1001.json', commentId, "/envelope/array-node('comments')"
    )
    if (res.success === true) context.getDocument('/person/1001.json'); //
Reload after delete
  }
  return (
    <div className="App">
      <div>
        <div style={{height: '240px'}}>
          <CommentBox
            label="Comments"
            inputPlaceholder="Add a comment"
            buttonLabel="Post"
            username="joe-user"
            imgSrc="https://demos.telerik.com/kendo-ui/content/web/Customers/
RICSU.jpg"
            profileImage={{
              alt: 'Avatar',
              path: 'https://demos.telerik.com/kendo-ui/content/web/
```

```
Customers/RICSU.jpg'
            }}
            onSubmit={(comment) => onSubmitComment(comment)}
          />
          <CommentList
                        data={context.documentResponse}
                        config={{comments: {
                                path: 'data.envelope.comments'
                        }}
                        username="joe-user"
                        onSaveComment={(id, comment) => onEditComment(id,
comment)}
                        onDeleteComment={(id) => onDeleteComment(id)}
                        numToShow={3}
                        numToLoad={2}
      />
          />
        </div>
      </div>
    </div>
  );
}

export default App;
```

CommentList example rendering

The CommentList example code renders this:



CommentList API

Prop	Type	Description
data	any	Array of comments to display.
config	object	A comments configuration object.
config.comments	PathConfig	The `comments.path`value is a JSONPath to the comments in the `data` prop.
username	string	Username of the current user. For testing, whether the current user is allowed to edit or delete a comment.
headerClassName	string	Class name applied to the header.
headerValue	ReactNode	Optional header for the comment list.
footerClassName	string	Class name applied to the footer.
footerValue	ReactNode	Optional footer for the comment list.
commentContainerStyle	CSSProperties	CSS styles applied to the widget container.
userInfoContainerStyle	CSSProperties	CSS styles applied to the user info container.
imgStyle	CSSProperties	CSS styles applied to the comment image.

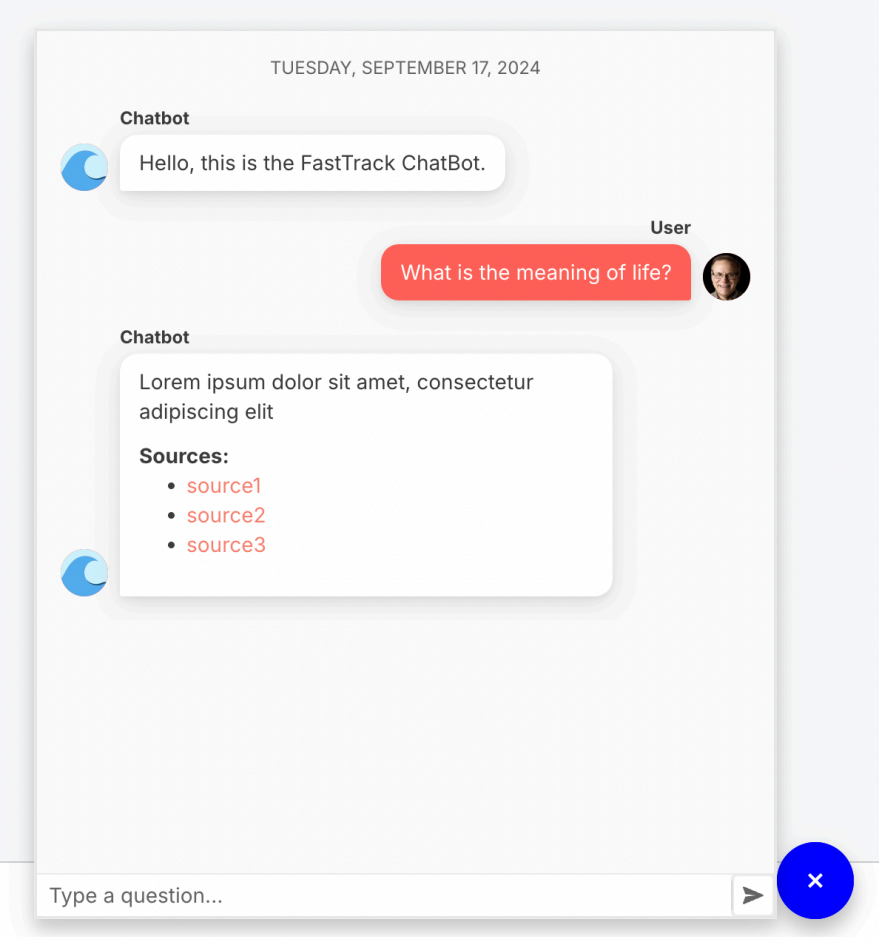
Prop	Type	Description
usernameStyle	CSSProperties	CSS styles applied to the username.
timestampStyle	CSSProperties	CSS styles applied to the timestamp.
contentStyle	CSSProperties	CSS styles applied to the comment content.
buttonsContainerStyle	CSSProperties	CSS styles applied to the button's container.
editLabel	string	Label for the edit button. The default label is **Edit**.
editStyle	CSSProperties	CSS styles applied to the edit button.
editPlaceholder	string	Placeholder text for the edit input field.
deleteLabel	string	Label for the delete button. The default label is **Delete**.
deleteStyle	CSSProperties	CSS styles applied to the delete button.
renderImage	ReactNode	Optional custom React component for rendering an image next to each comment.
saveLabel	string	Label for the Save button. The default label is **Save**.
cancelLabel	string	Label for the Cancel button. The

Prop	Type	Description
		default label is **Cancel**.
numToShow	number	Number of comments to show by default. If this is undefined, all the comments are shown.
numToLoad	number	Number of comments to load with the load button. The default is **2**.
loadLabel	string	Label for the load button. The default label is **Load more comments**.
loadStyle	CSSProperties	CSS styles applied to the **Load more comments** button.
summaryStyle	CSSProperties	CSS styles applied to the button comment summary.
formatTimestamp	function	Callback function to format timestamps. The function receives the comment timestamp as an argument and returns the formatted timestamp.
onSaveComment	(commentId: string, comment: { content: string, ts: string, username: string, imgSrc: string }, onSave: (id: string) => void) => void	Callback function triggered when saving a comment. Receives three parameters: the `commentId`, the comment object, and the `onSave`callback function. The `onSave` callback should be executed to update the widget after a comment is saved.
onDeleteComment	(commentId: string, onDelete: (id: string) => void) => void	Callback function triggered when deleting a comment. Receives two parameters: the commentId, and the `onDelete` callback function. The

Prop	Type	Description
		`onDelete` callback should be executed to update the widget after a comment is deleted.

ChatBot

The FastTrack ChatBot widget displays an interactive console that lets a user participate in a chat session with a large language model (LLM) by accepting text prompts and returning natural human-like responses to those prompts. By connecting the ChatBot to a backend that leverages the AI capabilities of MarkLogic RAG (Retrieval Augmented Generation), it receives contextually relevant responses with fewer inaccuracies that may contribute to data biases.



Implementing a Chatbot in a React Application

The ChatBot widget utilizes the KendoReact Conversational UI component to display the interactive chat console. The widget API represents the user and chatbot in various ways, manages how messages are sent and received, and formats the enclosing conversation container. The example code shows a React application with a configured ChatBot widget performing message handling:

```
import { useState } from "react";
import { ChatBot } from "ml-fasttrack"
function App() {
  const bot = {
    id: 0,
    name: "ExampleBot",
    avatarUrl: "/bot-icon.jpg"
  }
  const user = {
    id: 1,
    name: "User",
    avatarUrl: "/user-icon.jpg"
  }
  const initialMessages = [
    {
      author: bot,
      timestamp: new Date(),
      text: "Hello, ask me anything",
    }
  ];
  const [messages, setMessages] = useState(initialMessages);
  return (
    <>
      <ChatBot
        bot={bot}
        user={user}
        messages={messages}
        placeholder="Type a question"
        noResponseText="No response available"
        onSourceClick={(event, srcId) => console.log("Clicked", srcId)}
        parameters={{
          host: '127.0.0.1',
          port: 4015,
          path: '/api/fasttrack/chat',
        }}
      />
    </>
  )
}
```

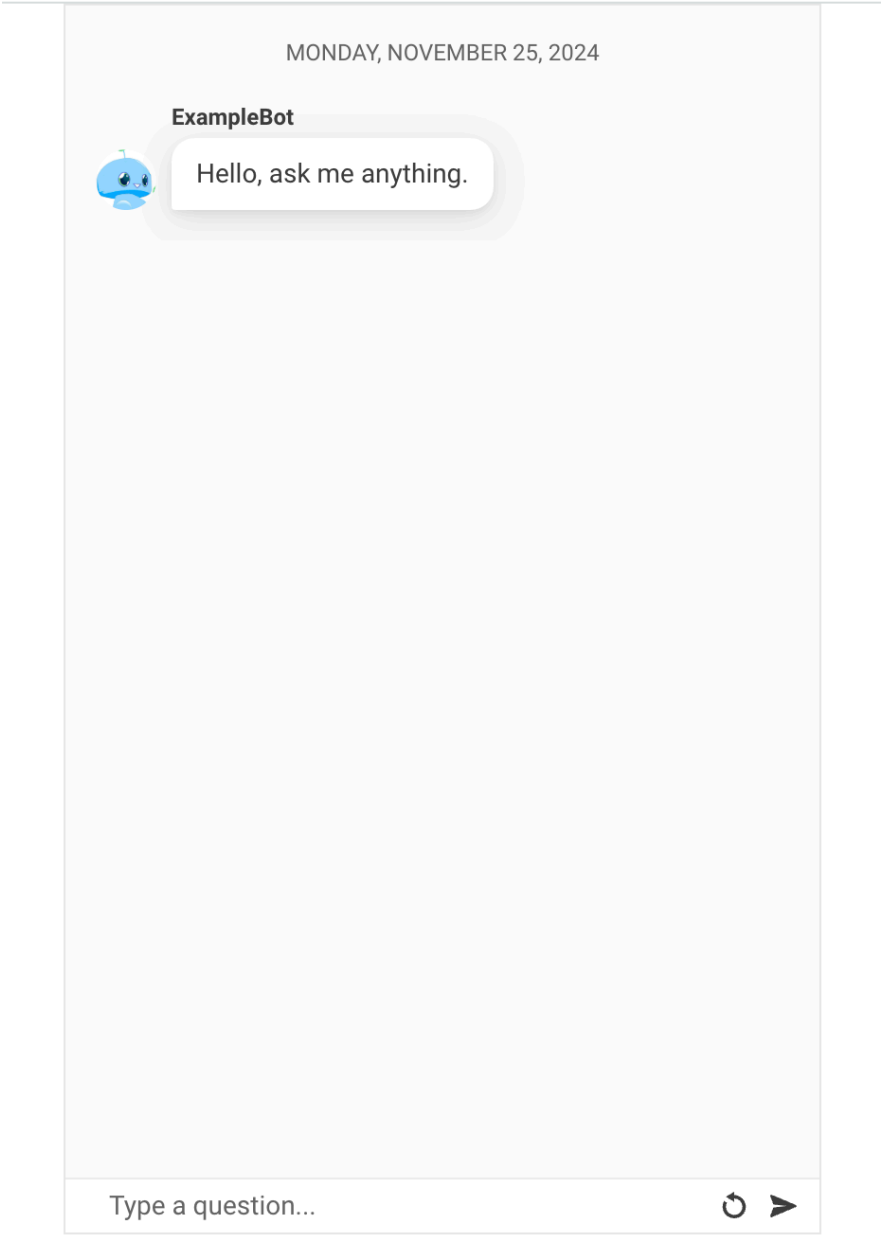
In the example above:

- The `bot` and `user` objects configure the bot and user outputs.
- A `messages` array holds the conversation messages and initializes with an introductory message from

the bot.

- The `onSourceClick` callback executes when a source item from the response is clicked.
- The `parameters` object defines how the UI connects to the chat backend. If `parameters` is not defined, the connection properties set in MarkLogicProvider are used with a `path` value of `api/fasttrack/chat`.
- The ChatBot widget uses the `postChat()` method in MarkLogicContext to handle messages.

These widget props are optional and default to plain configurations if none is provided. The example code output:



Implementing ChatBot with Custom Message Handling

```
import { useState, useContext } from "react";
import { ChatBot, MarkLogicContext } from "ml-fasttrack"
function App() {
  const bot = {
    id: 0,
    name: "ExampleBot",
    avatarUrl: "/bot-icon.jpg"
  }
  const user = {
    id: 1,
    name: "User",
    avatarUrl: "/user-icon.jpg"
  }
  const initialMessages = [
    {
      author: {bot},
      timestamp: new Date(),
      text: "Hello, ask me anything",
    }
  ];
  const context = useContext(MarkLogicContext);
  const [messages, setMessages] = useState(initialMessages);
  const addNewMessage = async (event) => {
    const latestUserMessage = event.message;
    // Add user message and then show loading indicator
    setMessages((oldMessages) => [...oldMessages, event.message]);
    setMessages((oldMessages) => [...oldMessages,
      {
        author: bot,
        typing: true,
      }
    ]);
    await context.postChat(latestUserMessage.text, context.combinedQuery, {})
    .then((response) => {
      setMessages((oldMessages) => [...oldMessages,
        {
          author: bot,
          timestamp: new Date(),
          // Remove any bracketed text from the response
          text: response.output.replace(/ *\[[^)]*\] */g, ""),
        }
      ]
    )});
  };
  return (
    <>
      <ChatBot
        bot={bot}
        user={user}
```

```
        messages={messages}
        onMessageSend={addNewMessage}
        customBotResponse={true}
      />
    </>
  )
}
export default App
```

In the example above:

- The callback function `addNewMessage`:
  - Receives an event object containing the message from the user.
  - Adds message text from the user to the conversation console and displays a typing indicator.
  - Executes the `postChat()` method from MarkLogicContext to retrieve a response to the error message.
  - Creates a new message object when a response is returned and adds to the `messages` array. The new message then appears in the conversation console. In this example, the message is transformed to remove any bracketed text.
- The `customBotResponse` method is set to true to prevent the ChatBot widget from adding extra messages to the messages array.

KendoReact Props

ChatProps

Prop	Type	Description
user	`User`	Represents the user participant in the chat. See the KendoReact UserProps.
bot	`User`	Represents the bot participant in the chat. See the KendoReact UserProps.
messages	`Message[]`	Represents an array of Chat Messages. See the KendoReact MessageProps.
timestampFormat	`string`	Format for the timestamp of the chat messages. The default is `M/d/y h:mm:ss a`.

Prop	Type	Description
placeholder	`string`	Placeholder text for the message input field when it is empty.
width	`stringnumber`	Sets the width of the ChatBot. The default is `""`.
responseTransform	`function`	Callback function for transforming the chat response. Accepts the chat response and returns a transformed response to be used by the widget.
messageBox	`ComponentType<ChatMessageBoxProps>`	Custom component to render the message input field and the Send button. See the KendoReact ChatMessageBoxProps.
onMessageSend	`((event: any) => void)`	Callback function triggered when the user types a message and clicks the Send button or presses Enter.
onSourceClick	`((event: any, sourceId: any) => void)`	Callback function triggered when a source is clicked. The function can receive the event and respective source ID as arguments.
sourceTruncation	`number`	Number of characters to display for each source item label. If the source label text exceeds the truncation value, it is truncated and an ellipsis is added.
customBotResponse	`booleanFalse`	Boolean value which controls whether to use the widget's default bot response or not. When set to true, create a custom method to update the 'messages' object with your ideal bot response . The default is `false`.
noResponseText	`string`	Text to display when the bot has no response to the user message. This is displayed as a chat message.

Prop	Type	Description
className	`string`	Class name applied to the widget.
toolbar	`any`	Renders any component to be displayed as a toolbar inside the chat.
showToolbar	`booleanFalse`	Whether the toolbar is rendered. The default is `false`.
showRestart	`booleanFalse`	Whether the restart button to clear the conversation is rendered. The default is `false` .
showSources	`booleanTrue`	Whether source citations for bot responses are displayed in the chat, assuming `customBotResponse` is set to `false`.
settings	`Record<string, any>`	Additional KendoReact ChatProps to pass into the component.
parameters	`{ scheme?: string; host?: string; port?: number | undefined; path?: string | undefined; auth?: { username?: string | undefined; password?: string | undefined; } | undefined; } | undefined`	An object of parameters for connecting to the ChatBot service. These will override the connection settings defined with MarkLogicProvider.

DataGrid

The DataGrid widget is based on the KendoReact Data Grid component and displays content from `/v1/search` results in a table. Optional pagination controls allow users to navigate the pages in the results.

DataGrid MarkLogic setup

Content from documents in search results can be displayed in a DataGrid table. The content must be extracted and included in search results. To do this, the `extract-document-data` property can be used in the query options of the application. See Include document extracts in search results for details.

Display search results in a table

In addition to displaying content from documents, the DataGrid widget can display responses from a `/v1/search` in a table. Content for each result in the search response is displayed in a table row. For example, consider this `/v1/search` response payload:


```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
                            "status": "active",
                            "activities": [
                                {
                                    "description": "Started at Fadeo",
                                    "ts": "2013-06-22"
                                },
                                {
                                    "description": "Promoted",
                                    "ts": "2019-08-15"
                                }
                            ]
                        }
                    }
                ]
            }
        },
```

```
        // more results...
    ],
    // more metadata...
}
```

Content from this payload can be displayed in a React application using this code:

```
import { useContext } from "react";
import "./App.css";
import { MarkLogicContext, SearchBox, DataGrid } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const handleClick = (uri) => {
    console.log("Result clicked: " + uri)
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div>
        <DataGrid
          data={context.searchResponse.results}
          gridColumns={
                [
              { title: "Index", field: "index" },
              { title: "URI", field: "uri" },
              {
                title: "First Name",
                cell: (props) =>
(<td>{(<span>{props?.dataItem?.extracted.content[0].envelope.firstName}</span
>)}</td>)
              },
              {
                title: "Last Name",
                cell: (props) =>
(<td>{(<span>{props?.dataItem?.extracted.content[0].envelope.lastName}</span>
)}</td>)
              },
              {
                title: "Actions",
                cell: (props) => (
                  <td
```

```
                    onClick={() => handleClick(props?.dataItem?.uri)}
                    style={{ color: '#0d6efd', cursor: 'pointer' }}
                  >
                    {(<span>{"Click me"}</span>)}
                  </td>
                )
              }
            ]
          }
          pagerButtonCount={5}
          pageSizeChanger={[1, 2, 5]}
          paginationSize="medium"
          paginationFooter={true}
          showPreviousNext={true}
          showInfoSummary={true}
        />
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In the example:

- The `data` prop is set to the results in the search response in the application context.
- The search results are populated from the queries submitted by the SearchBox widget.
- The table columns are configured with an array in the `gridColumns` prop.
- The `title` property in each `gridColumns` object defines the title of the column.
- The content in the columns for each result can be configured with a field property or cell callback function:
  - field property - use the property to define the path to a value at the result root. (For example, the index or uri of each search result).
  - cell callback function - this method displays values other than those at the result root. The callback receives an object argument with the `dataItem` property set to the result content. The callback must return content wrapped in a `td` tag to show a table cell.

DataGrid example rendering

The example code in Display search results in a table displays a table like this:

Index	URI	First Name	Last Name	Actions
1	/person/1001.json	Nerta	Hallwood	Click me
2	/person/1002.json	Shaylynn	Guard	Click me

1 2 2 items per page 1 - 2 of 3 items

DataGrid API

Property	Type	Description
data	object	Array of objects to display in the table.
gridColumns	GridColumnProps[]	Array of configuration objects that define each column in the table. See the KendoReact GridColumnProps.
initialSort	SortDescriptor[]	Configuration for the initial sorting. See the KendoReact SortDescriptor.
sortable	boolean	Allow sorting in the columns.
resizable	boolean	Allow resizable columns.
style	CSSProperties	CSS styles applied to the widget.
onPageChange	((event: GridPageChangeEvent) => void)	Callback function to handle paging. Overrides the native paging function. See the KendoReact GridPageChangeEvent.
onSortChange	((event: GridSortChangeEvent) => void)	Callback function to handle sorting. Overrides the native sort function.

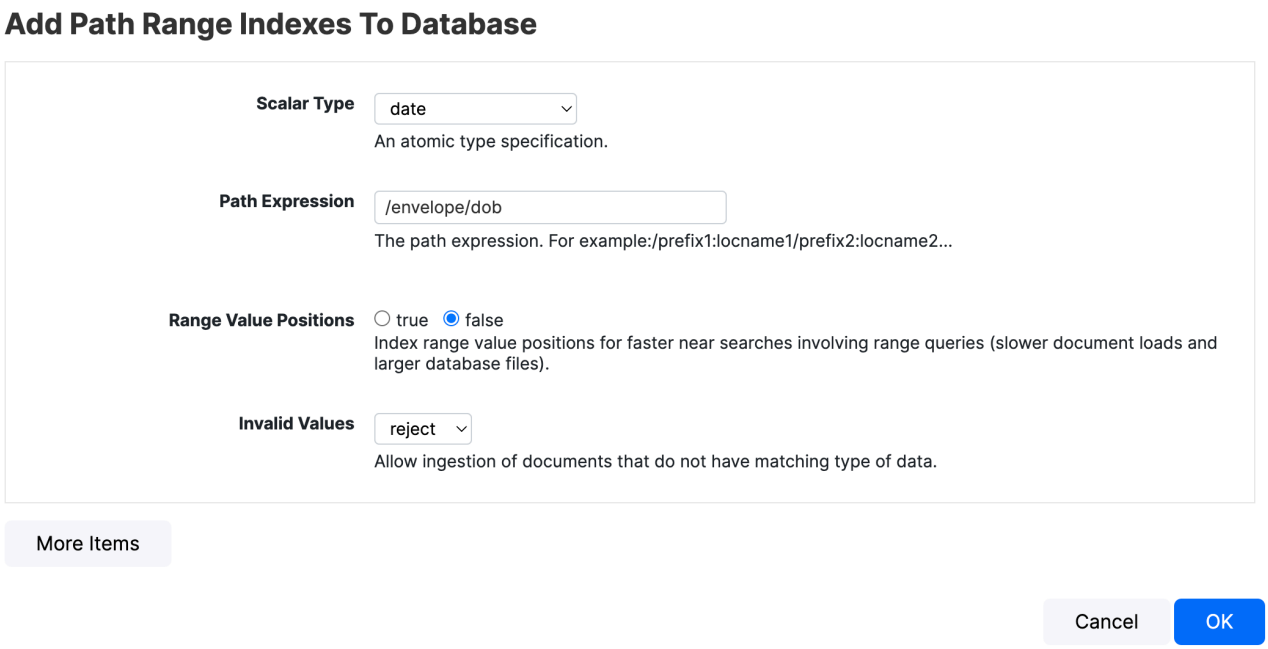
Property	Type	Description
		See the KendoReact GridSortChangeEvent.
paginationHeader	boolean	Indicates whether to display pagination controls in the header.
paginationFooter	boolean	Indicates whether to display pagination controls in the footer.
pageSizeChanger	number[] | (string | number)[]	Numeric array for displaying a menu in the pagination for configuring the number of items on each page.
showPreviousNext	boolean	Whether to display previous and next buttons in the pagination.
showInfoSummary	boolean	Optional value for displaying the results summary in the pagination options.
paginationClassName	string	Class name applied to the pagination container.
paginationSize	"small" | "medium" | "large"	Size of the pagination.
pagerButtonCount	number	The number of page buttons to display in the pagination controls.

DateRangeFacet

The DateRangeFacet widget displays start and end date pickers for a faceted date property in a set of search results.Users can select start and end values to constrain a search by the selected range.

DateRangeFacet MarkLogic setup

Faceted search in MarkLogic requires a range index on the faceted property. A range index can be added for a property using the Admin Interface or a MarkLogic API. This screen shot shows a path range index added to the `dob` property in the Admin Interface:



Once the index is added, facets for the property can be returned in search results. To do this, a constraint is added in the query options of the application:

```
<?xml version="1.0" encoding="UTF-8"?>
<options xmlns="http://marklogic.com/appservices/search">
   <constraint name="dob">
        <range type="xs:date" facet="true" collation="">
            <path-index>/envelope/dob</path-index>
        </range>
   </constraint>
   <return-facets>true</return-facets>
</options>
```

The `constraint` settings correspond to the settings for the index. `return-facets` must be set to `true` so that facet results are returned with the search results. This example shows the `dob` facet information returned in the search response:

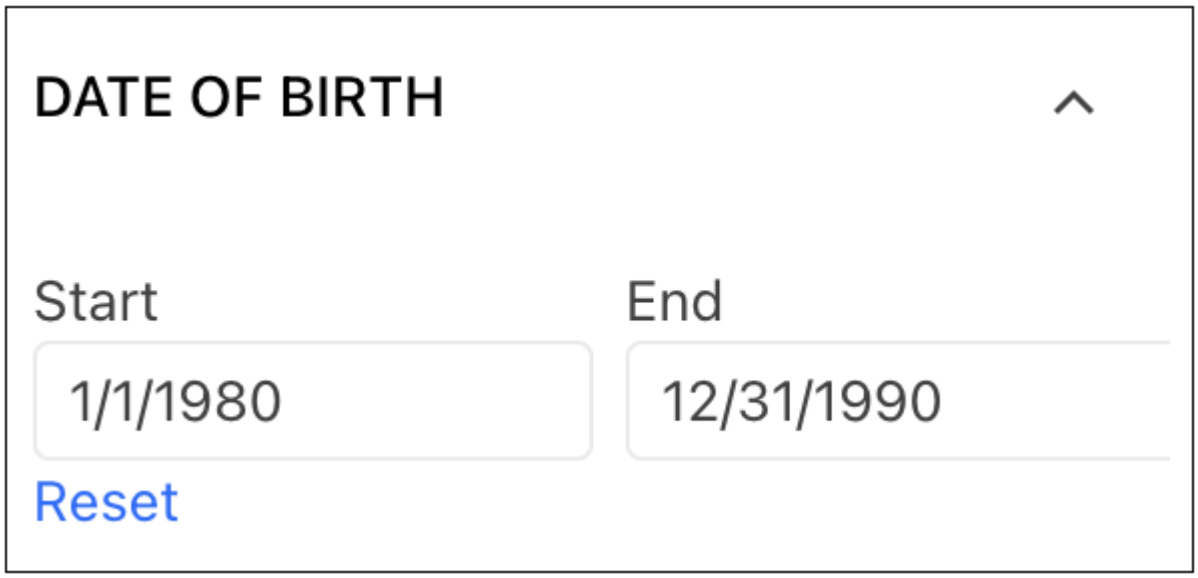
```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
```

```
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "score": 0,
            "confidence": 0,
            "fitness": 0,
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "mimetype": "application/json",
            "format": "json",
            "matches": [
                {
                    "path": "fn:doc(\"/person/1001.json\")/object-node()",
                    "match-text": [
                        "person Nerta Hallwood Marketing Manager Active
1985-03-04 104000 person-1001.jpg"
                    ]
                }
            ],
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
                            "status": "Active",
                            "dob": "1985-03-04",
                            "salary": 104000,
                            "image": "person-1001.jpg",
                        }
                    }
                ]
            }
        },
        // ...
    ],
    "facets": {
        "dob": {
            "type": "xs:date",
            "facetValues": [
                {
                    "name": "1964-09-30",
                    "count": 1,
                    "value": "1964-09-30"
                },
                {
```

```
                    "name": "1985-03-04",
                    "count": 1,
                    "value": "1985-03-04"
                },
                {
                    "name": "1988-12-15",
                    "count": 1,
                    "value": "1988-12-15"
                }
            ]
        }
    },
    // ...
}
```

DateRangeFacet example rendering

This example shows the DateRangeFacet widget. The widget has start and end date pickers for a faceted `dob` property in the results. With the constraint applied, only documents where the `dob` property is between 1/1/1980 and 12/31/1990 are returned and displayed.



DateRangeFacet example configuration

This code shows the DateRangeFacet widget imported and configured in a React application:

```
import { useContext, useState } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsSnippet, DateRangeFacet } from
"ml-fasttrack";
```

```
function App() {

  const context = useContext(MarkLogicContext);

  const [dateVals, setDateVals] = useState({ start: new Date(1980, 0, 1),
end: new Date(1990, 11, 31) });

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const updateDateRange = (constraint, previousConstraint, event) => {
    constraint && context.addRangeFacetConstraint(constraint)
    constraint === undefined &&
context.removeRangeFacetConstraint(previousConstraint)
    setDateVals(event?.value)
  }

  const resetDateRange = (event, dateVals, constraint) => {
    context.removeRangeFacetConstraint(constraint)
    setDateVals({ start: new Date(1980, 0, 1), end: new Date(1990, 11, 31) })
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div style={{display: 'flex', flexDirection: 'row'}}>
          <div style={{width: '640px'}}>
            <ResultsSnippet
              results={context.searchResponse.results}
              paginationFooter={true}
            />
          </div>
          <div>
            {context?.searchResponse?.facets?.dob &&
              <DateRangeFacet
                title={'Date of Birth'}
                name={'dob'}
                isFacet={true}
                value={dateVals}
                onSelect={updateDateRange}
                resetVisible={true}
                onReset={resetDateRange}
              />
            }
          </div>
      </div>
    </div>
  );
```

```
}

export default App;
```

DateRangeFacet code explanation

In the DateRangeFacet example configuration:

- The `name` prop is set to the name of the date facet to display from the search response object.
- The selection information from the widget is updated in the `dateVals` state variable by the `onSelect` callback. This information is cleared by the `onReset` callback.

DateRangeFacet API

Prop	Type	Description
title	string	Facet title for the collapsible header when in facet mode and `showAccordion` is not false.
subTitle	string	Facet subtitle for the collapsible header when in facet mode and `showAccordion` is not false.
name	string	Facet name in facet data.
defaultValue	object	Sets the default value of the DateRangeFacet. See the Kendo DateRangeFacetProps. For example: `{ start: new Date(1960, 0, 1), end: new Date(1990, 11, 31) }`.
dateFormat	string	Specifies the date format. See the date-fns documentation. The default value is `"yyyy-MM-dd"`.

Prop	Type	Description
options	string	Sets the date-fn options for the "format(date, format, options)" function. See the date-fns documentation.
showAccordion	boolean	Indicates whether to show the expansion header.
containerStyle	CSSProperties	CSS styles applied to the widget.
startOperator	string	Sets the operator for the range start. See the MarkLogic documentation. The default is `GE`.
endOperator	string	Sets the operator for the range end. See the MarkLogic documentation. The default value is `LE`.
resetText	string	Text of the reset button. The default value is `Reset`.
resetClassName	string	CSS class name to apply to the Reset button.
reset	ReactNode	React element for defining a custom Reset button.
onSelect	`((value: { type: string; name: string; value: string; operator: string; title: string; }[], previousValue: any, e: any) => void)`	Callback function for the select event. Receives an array of selection objects for the start and end date values, the previous value, and the event.
onReset	`((value: any, facetValue:`	Callback function for the reset event.

Prop	Type	Description
	`any, e: any) => void)`	Is passed the current value, facet value, and event.

DateRangeFacet callbacks

Example date range object passed to callbacks

```
{
    "start": "1980-01-01T08:00:00.000Z",
    "end": "1990-12-31T08:00:00.000Z"
}
```

Example constraint passed to callbacks

```
[
    {
        "type": "date",
        "name": "dob",
        "value": "1980-01-01",
        "operator": "GE",
        "title": "Date of Birth"
    },
    {
        "type": "date",
        "name": "dob",
        "value": "1990-12-31",
        "operator": "LE",
        "title": "Date of Birth"
    }
]
```

EntityRecord

The EntityRecord widget organizes properties from a record into a UI container. Props control styling and functionality. EntityRecord works as a standalone container on a page or within another component, such as the WindowCard widget. To highlight key properties from records in a list of search results, EntityRecord can also be tied to search-result click events.

EntityRecord example configuration

In this example, the EntityRecord widget displays the data properties of a document returned from MarkLogic using `MarkLogicContext` and a `useEffect` hook:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, EntityRecord } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  useEffect(() => {
    context.getDocument('/person/1001.json');
  }, []);

  const entityRecordConfig = {
    entityTypeConfig: {
      "path": "data.*~"
    },
    entities: [
      {
        entityType: 'person',
        entityTypeDisplay: 'Person',
        title: {
          path:  'data.person.fullname',
        },
        items: [
          {
            label: 'STREET',
            path: 'data.person.address.street'
          },
          {
            label: 'CITY',
            path: 'data.person.address.city'
          },
          {
            label: 'STATE',
            path: 'data.person.address.state'
          },
          {
            label: 'COUNTRY',
            path: 'data.person.address.country'
          },
          {
            label: 'DOB',
            path: 'data.person.dob'
          },
          {
            label: 'CONTACT',
```

```
            path: 'data.person.contacts.contact[0].value'
          }
        ],
        avatarProps: {
          border: false,
          themeColor: 'info',
          rounded: 'full',
          type: 'image',
          style: { flexBasis: 140, height: 140 },
          avatarImage: {
            path: 'data.person.images.image[0].url',
            alt: 'person image'
          },
        }
      }
    ]
  }

  const handleClick = (attributes, _event) => {
    const uri = attributes?.uri;
    if (uri) {
      console.log('URI: ' + uri)
    }
  }

  return (
    <div className="App">
      <div>
        <EntityRecord
            recordActionLabel={'uri'}
            entity={context.documentResponse}
            config={entityRecordConfig}
            badges={[
        {
          label: 'graph',
          color: 'primary',
          onClick: () => console.log('badge clicked!')
        }
      ]}}
            onRecordActionClick={handleClick} />
      </div>
    </div>
  );
}
```

Code explanation

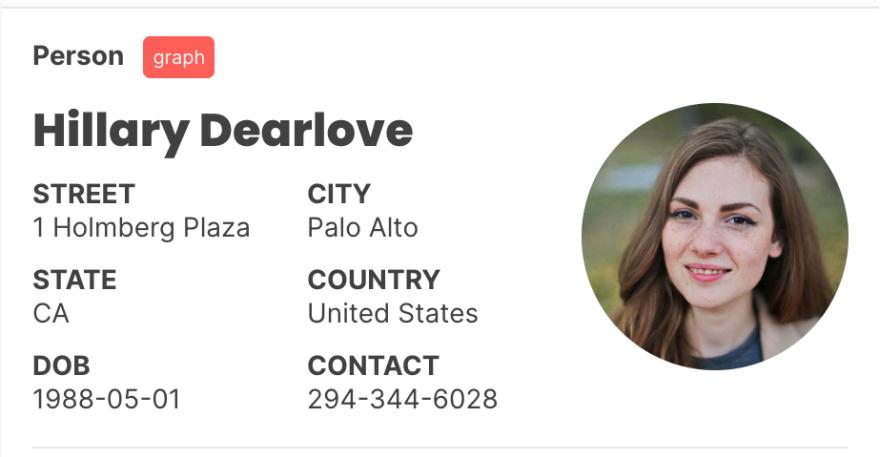
In the EntityRecord example configuration:

- The `entity` prop accepts a JSON object of the returned document.

- The `entityTypeConfig` prop accepts an object that determines where in the record the entity type is defined.
- The `entities` prop accepts an array of entity configuration objects. Each object determines what data from the record is displayed in the widget for a given entity type.
- The widget can display one or more action badges. The `badges` prop accepts an array of badge configuration objects.

EntityRecord example rendering

This is an example rendering of the EntityRecord widget:



EntityRecord API

Prop	Type	Description
entity	object	Entity data to display.
config	object	Array of entity configuration objects. Each object determines what data to display in a result for an entity type. See the EntityRecord config API.
itemsContainerStyle	CSSProperties	CSS styles applied to the item's container.

Prop	Type	Description
itemContainerStyle	CSSProperties	CSS styles applied to each item.
itemLabelStyle	CSSProperties	CSS styles applied to the item labels.
itemValueStyle	CSSProperties	CSS styles applied to the item values.
multipleValueSeparator	string	Symbol that separates multiple values when an item is an array. Deprecated: Use the `separator` property in PathConfig instead.
recordActionLabel	string	Label for the record action badge.
recordActionColor	string	KendoReact theme color for the record action badge. See the KendoReact BadgeProps.
recordActionStyle	CSSProperties	CSS styles applied to the record action badge.
onRecordActionClick	((entity: any, event: any) => void)	Callback function triggered by a click event on the record action button.

EntityRecord config API

Prop	Type	Description
entityTypeConfig	PathConfig	Entity type configuration object.
entityTypeConfig.path	string	Path to the entity type in the search result. The path is specified using `JSONPath`.

Prop	Type	Description
entities[]	object[]	Array of configuration objects for each entity type.
entities[].entityType	string	Entity type of the configuration object.
entities[].entityTypeDisplay	string	Label displayed for the entity type name.
entities[].title	PathConfig	Title configuration object.
entities[].title.path	string	Path to the title property in the entity data. The path is specified using `JSONPath`.
entities[].items[]	PathConfig[]	An array of configuration objects that display each property.
entities[].items[].label	string	Property label.
entities[].items[].path	string	Path to the property value. The path is specified using `JSONPath`.
entities[].image	PathConfig	Image configuration object.
entities[].image.path	string	Path to the url value for the image. The corresponding image will be rendered. The path is specified using `JSONPath`.
entities[].avatarProps	AvatarProps	An avatar object for an image to be displayed.

ExportData

The ExportData widget displays a button for downloading application content. When configuring the widget, the value to be exported can be data, a function that returns data, or a Promise that resolves to data.

By default, the widget offers three format options: comma-separated values (CSV), JSON, and XML. When you click the button, a dropdown menu appears allowing the user to select a format. After a format is selected, the value is exported. The widget can also be configured to export custom formats.

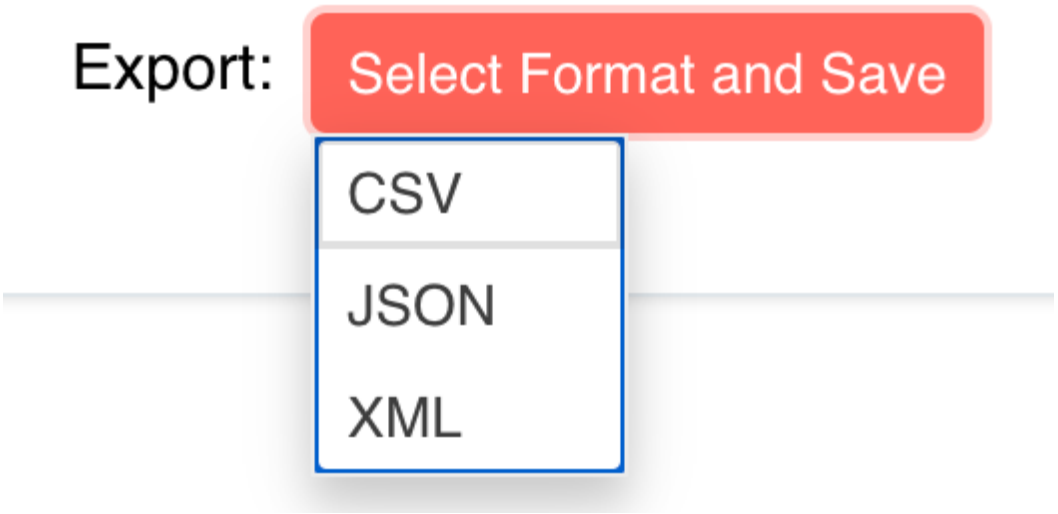
If the widget is configured to export only one type of format, no dropdown is displayed and the export occurs on button click.

Configuration with Default Formats

```
import { ExportData } from "ml-fasttrack";
// ...
<ExportData
  data={context.searchResponse.results}
  prefix={'Export:'}
  label={'Select Format and Save'}
  themeColor={'primary'}
/>
```

The above code displays the widget with the three default format options.

Rendered Widget



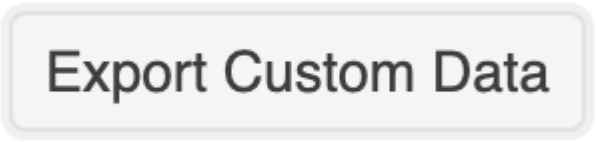
Configuration with Custom Export

```
import { ExportData } from "ml-fasttrack";
// ...
const customExport = (data) => {
  return JSON.stringify({ envelope: data })
}
<ExportData
```

```
  data={myData}
  label={'Export Custom Data'}
  formats=[{
    text: 'Custom Format',
    id: 'myFormat',
    extension: '.json',
    transform: customExport
  }]
  filename='custom-export'
  appendTimestamp={false}
/>
```

The above code exports data as a custom JSON object. The exported content is saved with the filename "custom-export.json".

Rendered Widget



API

Prop	Type	Description
data	any	Content to export. Can be data, a function that returns data, or a Promise that resolves to data.
formats[]	object[]	An array of format configuration objects.
formats[].text	string	The text label for the dropdown menu.
formats[].id	string | number	The ID for the format.
format[].extension	string	The filename extension for the exported file.

Prop	Type	Description
format[].transform	function	The function that transforms the data to the exported content. Receives the `data` value and returns the value to export.
formatsShown	string[]	Array of IDs of formats to show in the menu in display order. This allows you to limit the default formats shown or reorder the formats.
className	string	Class name applied to the widget.
prefix	string	Prefix string to display to the left of the button.
label	string	Label to apply to the button.
themeColor	string	KendoReact theme color to apply to the button. See the KendoReact ButtonProps.
settings	any	Additional KendoReact DropDownButton or Button props to apply to button.
filename	string	Text to use as the exported filename.
appendTimestamp	boolean	Whether to append an underscore followed by a timestamp the exported filename.

GeoMap

The GeoMap widget leverages ESRI's ArcGIS to display data containing real-world location information on an interactive map.

ESRI integration

No additional installation steps are needed to use the GeoMap with ESRI once FastTrack is installed. However, developers will need to acquire their own ArcGIS developer API Key (see Introduction to API key authentication) and add it to the GeoMap configuration. This can be accomplished by setting the `esriApiKey` prop on GeoMap to the developer's ESRI API Key.

Markers configuration

Data on the GeoMap can be visualized with markers using the `"markers"` and `"transformMarkers"` props.

The `"markers"` property should be set to the collection of results data to visualize on the map. The results format should be an array of objects with latitude and longitude properties.

The `"transformMarkers"` property should be set to a function that transforms the data into the proper GeoMap format with latitude and longitude properties. This transform function should iterate over the results and return each data object in this format:

```
{
        point: {
            latitude: yourLatitude
        longitude: yourLongitude
        uri: yourURI
    }
}
```

To style each respective data point, add a symbol object inside the returned object in the transform. The symbol object can contain type, color, text, and font customizations as shown in the code.

> **Note:**
>
> Text values should correspond to ESRI standardized icon font text See Esri Icon Font (Calcite theme).

```
{
        point: {
            latitude: yourLatitude
        longitude: yourLongitude
        uri: yourURI
    },
        symbol: {
        type: "text",
    color: "Tomato"
```

```
    text: "\ue61d", // esri-icon-map-pin
    font: {
            size: 24
        }
}
```

Shapes configuration

Static shapes can be placed on the GeoMap in a similar fashion using the `"shapes"` and `"transformShapes"` props.

These static shape renderings should not be confused with the "sketch" functionality that enables polygon drawing on the map to narrow down search results.

The `"shapes"` property should be set to an array of objects with each object containing a geometry object and a styling object for each respective shape. The geometry object should consist of the shape type and `"rings"` array of vertices making up that shape, or the `"paths"` array for lines. For example:

```
geometry: {
      type: "polygon",
      // Bermuda Triangle
      rings: [
        [
          [-64.78, 32.3],
          [-66.07, 18.45],
          [-80.21, 25.78],
          [-64.78, 32.3]
        ]
      ]
    },
    symbol: {
      type: "simple-fill",
      color: [227, 139, 79, 0.4],
      outline: {
        type: "simple-line",
        color: [0, 128, 0],
        width: 1
      }
    }
```

```
geometry: {
      type: "polyline",
      paths: [
        [
          [-123.1207, 49.2827], // Vancouver
          [-114.0719, 51.0447], // Calgary
          [-113.4937, 53.5461]  // Edmonton
        ]
```

```
      ]
    },
    symbol: {
      type: 'simple-line',
      color: "blue",
      width: 3
    }
```

Geospatial search using polygons

This section contains database requirements and widget specifications.

Database requirements

For MarkLogic to identify geospatial properties in your data, geospatial indexes must be configured in the MarkLogic database. This can be done through the Admin UI (http://localhost:8001) or through MarkLogic's various programmatic APIs. See GeoMap API.

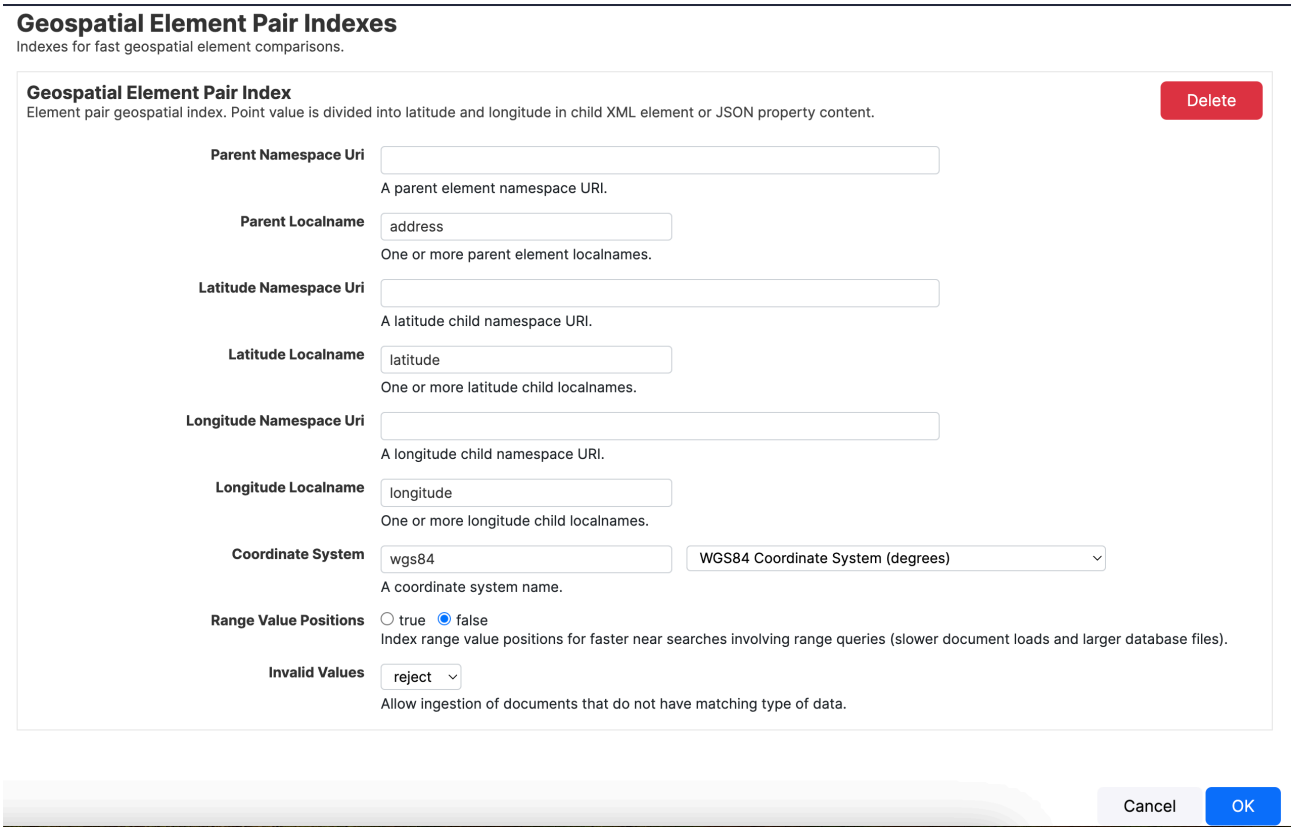
Admin UI

To configure a geospatial index using the Admin UI:

1. Access the content database that contains your data in the GUI.
2. The subsection **Geospatial Point Indexes** lets you configure various types of geospatial indexes. Which one you use will depend on how the latitude and longitude coordinates in the documents are structured.
3. For coordinates nested under a parent element, create a geospatial element pair index. Specify the following fields at a minimum:
    1. `"Parent Localname":` parent property that has the latitude and longitude values as children.
    2. `"latitude"`: child property with the latitude coordinate.
    3. `"longitude"`: child property with the longitude coordinate.

For more about MarkLogic geospatial indexes, see: Understanding Geospatial Query and Index Types.

Example configuration in the Admin UI



Rest API

Indexes can also be setup with a PUT call to MarkLogic's REST API `/manage/v2/databases/{id|name}/properties`.

> **Note:**
>
> Substitute the databases ID or name for `id|name`.

To setup a Geospatial Element Pair Index on a database using a JSON payload, pass this code in the HTTP body:

```
{
  "geospatial-element-pair-index": [
    {
      "parent-namespace-uri": "",
      "parent-localname": "PARENT-ELEMENT-NAME",
      "latitude-localname": "LATITUDE-ELEMENT-NAME",
```

```
      "latitude-namespace-uri": "",
      "longitude-localname": "LONGITUDE-ELEMENT-NAME",
      "longitude-namespace-uri": "",
      "coordinate-system": "wgs84",
      "range-value-positions": false,
      "invalid-values": "reject"
    }
  ]
}
```

For more about using the REST API for configuring databases, see: PUT /manage/v2/databases/{id|name}/properties.

Widget specifications

To use geospatial search with polygons, configure the `sketch`, `sketchPosition`, `selectionTools`, and `onGeoSearch` properties.

`sketch`

Set this property to `true` to enable polygon drawing capabilities on the map. This opens a sketch toolbar providing a selection of shapes to be drawn.

`sketchPosition`

Repositions the toolbar as desired. Possible values are `"top-right"`, `"top-left"`, `"bottom-left"`, and `"bottom-right"`.

`selectionTools`

Configures the shapes offered as selection options to sketch on the map. These will be displayed on the sketch toolbar. By default, all options are displayed. To disable an option, pass in an object that sets the `"polygon"`, `"circle"`, `"rectangle"`, `"rectangle-selection"`, or `"lasso-selection"` object to `false`. For example:

```
selectionTools={
    {
        'lasso-selection': false,
        'rectangle-selection': false
    }
}
```

`onGeoSearch`

This property takes in a callback function that should build and add a geo-constraint to the search query in MarkLogicContext based on coordinates of the polygon(s) drawn on the GeoMap. These respective polygon coordinates are provided by ESRI and automatically propagated from the GeoMap widget to the argument belonging in the specified callback function. Within this callback function, all specifications pertaining to the geospatial query being made should be defined respective to your data. The required parameters to set are:

1. type : string value describing the type of geospatial query. See Syntax Reference.
2. lat : string value matching the property in your data where `"latitude"` coordinates are defined.
3. lon : string value matching the property in your data where `"longitude"` coordinates are defined.
4. parent : string value matching the property of the object in your data where both latitude/longitude coordinates lie under.
5. polygon : Array values containing a transformation of all `"point"` objects that form the search polygon's vertices (see example transform below).

Once these parameters are defined in a query object, the final step is to add the geo-constraint to the MarkLogicContext which will automatically trigger a geo-spatial search:

```
const handleGeoSearch = (polygonsCoordsArr) => {
    const query =
        {
          "type": "geo-elem-pair-query",
          "lat": "Latitude",
          "lon": "Longitude",
          "parent": "Location",
          "polygon": polygonsCoordsArr.map((coords) => {
              const pointObj = {
                "point": coords[0]?.map((ele) => {
                  return {
                    "latitude": parseFloat(ele[1]),
                    "longitude": parseFloat(ele[0])
                  }
                })
              }
              return pointObj
          })
        }
    context.addGeoConstraint(query);
  }
```

2D and 3D maps

This section includes information on 2D and 3D maps.

2D map

1. To initialize the GeoMap as a 2D map, set the GeoMap props `"viewType"`, and `"useWeb"`.
2. (Optional) provide values for `"center"` and `"zoom"` to preset the default display.
    1. `"center"` takes in an array of latitude and longitude coordinates representing the center where the display will start.
    2. `"zoom"` takes in any integer from 0 to 23 in order of least greatest magnitude of zoom.

3. Set `"viewType"` to `"2D"` and `"useWeb"` to `false`.

Center and zoom example

```
"center" =  [-98.556, 39.810]
"zoom" = 4
```

Example 2D map display of geospatial search results with markers

3D map

1. To initialize the GeoMap as a 3D map, the GeoMap props `"viewType"` ,`"useWeb"`, and `"camera"` should be set.

2. Set `"viewType" = "3D"` and `"useWeb" = true`.

The `"camera"` property is used to determine the observation point from which the visible portion (or perspective) of the `SceneView` is determined. Options such as elevation, heading, tilt, etc. can be configured. A value in this property will override any `"center"` or `"zoom"` properties defined. For in-depth examples, see camera).

Example 3D map display of geospatial search results represented with markers

GeoMap API

Prop	Type	Description
basemap	string	ESRI basemap name. Default: `"streets-relief-vector"`.
center	[number, number]	Map center as a tuple of latitude and longitude values.
zoom	number	ESRI map zoom value. Usually ranges from 0 (global view) to 23 (detailed view).
markers	object[]	Array of marker definitions. `markers[].geometry` takes an object of Point properties, an mgrs property for MGRS values, or a utm property for UTM values. `markers[].symbol` takes an object of Symbol properties.
shapes	object[]	Array of shape definitions. `shapes[].geometry` takes an object of Polygon properties or Polyline properties with a type of `"polygon"` or `"polyline"`, respectively. `shapes[].symbol` takes an object of SimpleLineSymbol properties or SimpleFillSymbol properties with a type of `"simple-line"` or `"simple-fill"` respectively.

Prop	Type	Description
ground	string	ESRI ground property for displaying a 3D map. Default: `"world-elevation"`.
viewType	string	Map view type (`"2D"` or `"3D"`). Default :`"2D"`.
showToggle	boolean	Show the basemap toggle widget. Default: `false`.
toggleBasemap	string	Basemap to toggle (in addition to the default basemap). Default: `"gray-vector"`.
togglePosition	string	Toggle position. `Default: "bottom-right".`
addMarkerOnClick	boolean	Add points by clicking on the map. Default: `true`.
esriApiKey	string	ESRI Developer API Key value.
transformMarkers	function	Callback function for transforming the markers payload. Accepts the markers property payload and returns a transformed payload (array of marker definitions) to be used by the widget.
transformShapes	function	Callback function for transforming the shapes payload. Accepts the shapes property payload and returns a transformed payload (array of shape definitions) to be used by the widget.
sketch	boolean	Enable sketching tool interface on the map.

Prop	Type	Description
createTools	object	Enable/disable various geometry creation options in the sketching tool interface.
selectionTools	object	Enable/disable various selection options in the sketching tool interface.
sketchPosition	string	Placement of sketching tool interface on the map. Default: `"top-right"`.
symbol	object	Default ESRI symbol to use for map markers.
width	string	Width of widget container as a CSS width string (e.g., `"500px"`).
height	string	Height of widget container as a CSS height string (e.g., `"300px"`).
useWeb	boolean	Use a 2D WebMap or 3D WebScene, which are maps where configuration information is retrieved from an ArcGIS server. If `true`, must also supply `portalItemID`. Default: `false`.
portalItemID	string	ESRI portalItem ID for displaying a WebMap or WebScene.
camera	object	ESRI camera object for initializing the observation point in a 3D map.
showGallery	boolean	Show the basemap gallery widget.
galleryQuery	string	Gallery query for determining available basemaps. Default: `"World Basemaps for Developers" AND owner:esri`.

Prop	Type	Description
galleryPosition	string	Gallery position. Default: `"top-right"`.
activeSelection	string	ID of currently selected marker.
activeSelectionColor	string	Color to apply to selected marker.
activeSelectionZoom	number	Zoom to apply when marker is selected.
onClickMap	((attributes: any, event: any) => void)	Callback function triggered when the map area is clicked. Receives an object of attributes of the clicked area or element and an event object.
onGeoSearch	((arr: []) => void)	Callback function for submitting a geospatial search query. Based on polygon coordinates that are passed in as an array.
popupTemplate	object	Template to set generic properties of the popovers in the map.
popupFeatureLayer	object	Array of FeatureLayers for creating complex scenarios with popovers.

JsonView

The JsonView widget displays formatted JSON. The widget is based on the json-viewer npm library. Props control the formatting and functionality of the JSON. The JsonView widget can be used to:

- Confirm responses are returned as expected from the backend.
- Inspect response payload property paths to configure other widgets.

JsonView example rendering

The JsonView widget looks like this:

```
"envelope": { 7 Items
  "snippet-format": string "snippet"
  "total": int 1
  "start": int 1
  "page-length": int 10
  "results": [ 1 Items
    0: { 10 Items
      "index": int 1
      "uri": string "/person/1003.json"
      "path": string "fn:doc("/person/1003.json")"
      "score": int 20480
      "confidence": float 0.7111177
      "fitness": float 0.7111177
      "href": string "/v1/documents?uri=%2Fperson%2F1003.json"
      "mimetype": string "application/json"
      "format": string "json"
      "matches": [ 1 Items
        0: { 2 Items
          "path": string "fn:doc("/person/1003.json")/envelope/address/text( .
          "match-text": [ … ] 3 Items
        }
      ]
    }
  ]
  "qtext": string "artisan"
  "metrics": { 3 Items
    "query-resolution-time": string "PT0.004902S"
    "snippet-resolution-time": string "PT0.000936S"
    "total-time": string "PT0.006995S"
  }
}
```


JsonView example configuration

In this example, the JsonView widget uses `MarkLogicContext` to display a MarkLogic search response. The search response JSON object is defined in the `data` prop:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, JsonView } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
```

```
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div>
        <JsonView
            data={context.searchResponse}
            displayDataTypes
            displayObjectSize
            enableClipboard
            groupArraysAfterLength={100}
            indentWidth={4}
            maxHeight="580px"
            quotesOnKeys
            rootName="envelope"
          />
      </div>
    </div>
  );
}

export default App;
```

JSONView API

Prop	Type	Description
data	object	JSON data to display.
displayDataTypes	boolean	Indicates whether to show data types as prefixes to values. Default is **true**.
displayObjectSize	boolean	Indicates whether to show the size of objects and arrays. Default is true.
enableClipboard	boolean	Indicates whether to show clickable icons for copying object and array data. Default is true.

Prop	Type	Description
groupArraysAfterLength	number	Displays array values in groups based on the value. Groups are displayed with bracket notation and can be expanded and collapsed by clicking the brackets. Default is `100`.
indentWidth	number	Indent width for nested objects. Default is `4`.
maxHeight	string	Maximum height of the container. Content that exceeds this height will scroll. Default is `"500px"`.
quotesOnKeys	boolean	Indicates whether to Include quotes around keys (e.g. `"name":` vs. `name`). Default is `true`.
rootName	string	Name of the root node. Default is `"root"`.

MarkLogicContext

MarkLogicContext enables FastTrack applications to connect to the MarkLogic REST API. MarkLogicContext exposes methods for performing searches, retrieving documents, setting search constraints, and more. FastTrack UI widgets do not communicate with MarkLogic directly. Instead, widgets communicate by executing methods from MarkLogicContext.

MarkLogicContext saves the responses returned from MarkLogic in state variables. Data from MarkLogic can be displayed in FastTrack applications by accessing these state variables. MarkLogicContext methods also return HTTP responses, so the application can handle them directly (if desired).

MarkLogicContext leverages React context to make the methods and variables accessible throughout a FastTrack-enabled application.

MarkLogicProvider

MarkLogicProvider is a React context provider that gives FastTrack widgets access to MarkLogicContext. After wrapping a React application component with MarkLogicProvider (see the example), all child widgets in

the application can import MarkLogicContext and have access to MarkLogicContext methods and state.

MarkLogicProvider props define the connection settings for the MarkLogic backend. In a three-tier application, set the props to the connection settings for the middle-tier server of the application.

MarkLogicProvider API

Prop	Type	Description
scheme	string	HTTP scheme used when connecting to the backend. Default: "http".
host	string	Host used when connecting to the backend. Default: "localhost".
port	number	Port used when connecting to the backend. Default: 4000
options	string	Name of the installed MarkLogic query options used for searches.
auth	object	Authentication object.
auth.username	string	User name used to authenticate with the backend.
auth.password	string	Password used to authenticate with the backend.
paginationOptions	object	Pagination object.
paginationOptions.pageLength	number	Page length for searches. Default: 10.
requestInterceptor	function	Transformation function applied to all backend requests. For details, see the Axios documentation.
responseInterceptor	function	Transformation function applied to all backend responses. For details, see the Axios documentation.

Prop	Type	Description
searchTransform	string	Name of the transformation installed on MarkLogic to apply to search results.
documentTransform	string	Name of the transformation installed on MarkLogic to apply to retrieved documents.
updateSearchOnChange	boolean	Whether to automatically execute `postSearch()` when search-related state variables in MarkLogicContext change. State variables that will trigger execution: `qtext`,`collections`,`directories`,`stringFacetConstraints`,`rangeFacetConstraints`,`geoConstraints`,`wordQueryConstraints` Default:`false`
debug	boolean	Whether to show FastTrack debugging messages in the console. Default: `false`

MarkLogicProvider example

```
import { MarkLogicProvider } from "ml-fasttrack"
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <MarkLogicProvider
    scheme="http"
    host="localhost"
    port="4001"
    options="my-options"
    auth={{ username: "theUser", password: "p4ssw0rd" }}
    paginationOptions={{ pageLength: 20 }}
    debug={true}
  >
    <App />
  </MarkLogicProvider>
);
```

MarkLogicContext methods

This section describes the MarkLogicContext methods.

setQtext(qtext)

This method sets the `qtext` search state property.

Argument	Type	Description
qtext	string	A string query.

When `qtext` changes, MarkLogicContext executes a new `/v1/search` using the current context state and populates the `searchResponse` property with the response.

setQtext(qtext) example

```
context.setQtext("fast OR track");
```

setCollections(collections)

This method sets the `collections` state property. Collections allow documents to be organized into subsets. Searches can then be constrained by one or more collections. When `collections` changes, MarkLogicContext executes a new search using the current context state and populates the `searchResponse` property with the response.

Argument	Type	Description
collections	string[]	An array of collections.

setCollections(collections) example

```
context.setCollections(["person", "organization"]);
```

setDirectories(directories)

This method sets the `directories` state property. Directories organize documents into subsets based on paths in URIs. Searches can be constrained by directories. When the `directories` state property changes, MarkLogicContext executes a new search using the current context state and populates the `searchResponse` property with the response.

Argument	Type	Description
directories	string[]	An array of directories.

setDirectories(directories) example

```
context.setDirectories(["/pets/dogs/", "/pets/cats/"]);
```

setPageStart(index)

This method sets the index of the first result to return when executing `postSearch()`. It sets the `start` parameter for POST /v1/search.

Argument	Type	Description
index	number	The index of the first result to return when executing `postSearch()`. Default: 1.

setPageStart(index) example

```
context.setPageStart(2);
```

setPageLength(length)

This method sets the number of results returned when executing `postSearch()`.

Argument	Type	Description
length	number	Page length when executing `postSearch()`. Default: 10.

setPageLength(length) example

```
context.setPageLength(20);
```

setOptions(name)

This method names a query options previously created via the `/v1/config/query` service. It sets the `options` parameter for POST /v1/search.

Argument	Type	Description
name	string	The name of the installed query options.

setOptions(name) example

```
context.setOptions("myQueryOptions")
```

setSearchTransform(name)

This method names a search result transformation previously installed via the `/config/transforms`service. It sets the `transform` parameter for POST/v1/search. If a search response is returned, the transformation is applied to the search response.

Argument	Type	Description
name	string	The name of the installed search response transformation.

setSearchTransform(name) example

```
context.setSearchTransform("myTransformFunction")
```

setDocumentTransform(name)

This method names a content transformation previously installed via the `/config/transforms` service. It sets the `transform` parameter for GET /v1/documents. The service applies the transformation to the document prior to constructing the response.

Argument	Type	Description
name	string	The name of the installed content transformation.

setDocumentTransform(name) example

```
context.setDocumentTransform("myTransformFunction")
```

postSearch(combinedQuery, parameters, config)

This method executes a call to `POST /v1/search`. The call returns a Promise. The Promise is fulfilled with an HTTP response when it is successful. The `searchResponse` state variable is also populated with the response.

Argument	Type	Description
combinedQuery	object	A combined query object.
parameters	object	Object of key/value pairs corresponding to URL parameters. For details, see POST /v1/search.
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

postSearch(CombinedQuery, parameters) example

This example searches for documents containing the strings "fast" or "track". The example returns results, starting with the second result.

```
const combinedQuery = {
  qtext: "fast OR track"
}

context.postSearch(combinedQuery, {start: 2}).then(response => {
  console.log(response);
})
```

updateSearch()

This method triggers a call to `postSearch()` using the search-related state variables in MarkLogicContext and the parameters defined in MarkLogicProvider. Use this method to manually trigger a search when `updateSearchOnChange` in MarkLogicProvider is set to `false`.

updateSearch() example

The following will trigger a call to `postSearch()` using the `qtext` that value has been defined. Because `updateSearchOnChange` is set to `false`, the `setQtext()` method does not automatically trigger a search.

```
<MarkLogicProvider
  host={host}
  port="4000"
  updateSearchOnChange={false}
>
  <App />
</MarkLogicProvider>

// ...

context.setQtext("fast OR track"); // Does not trigger search
context.updateSearch();
```

getSuggestedSearch(partialQ, limit, config)

This method executes a call to GET /v1/suggest. The call returns a Promise. When the response is successful, the promise is fulfilled with an HTTP response. The `suggestSearchResponse` state variable is also populated with the response. For more details, see Generating Search Term Completion Suggestions.

Argument	Type	Description
partialQ	string	A partial query string.
limit	number	The maximum number of selections to return. Default: 10.
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

getSuggestedSearch(partialQ, limit) example

```
context.getSuggestedSearch("New", 5).then(response => {
  console.log(response);
})
```

getDocument(uri, parameters, config)

This method executes a call to GET /v1/documents. The method returns a Promise. When the call is successful, the promise is fulfilled with an HTTP response. The document content will be in the `data` property of the response object. The `documentResponse` state variable will also be populated with the response.

Argument	Type	Description
uri	string	URI of the document to retrieve.
parameters	object	Object of key/value pairs corresponding to URL parameters. For details, see GET /v1/documents.
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

getDocument(uri, parameters) example

```
const uri = "/person/1001.json";

context.getDocument(uri, {database: "my-app-db"}).then(response => {
  console.log(response);
})
```

getSparql(query, parameters, config)

The `getSparql(query, parameters)` method executes a call to GET /v1/graphs/sparql. The next action depends on the results of the call:

- When the call is successful, a Promise is returned and the `sparqlResponse` state variable is populated with the response.
- If the call is unsuccessful, an error is written to the console.

Argument	Type	Description
query	string	SPARQL query.
parameters	object	Object of key/value pairs corresponding to URL parameters. For details, see GET /v1/graphs/sparql.
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

getSparql(query, parameters) example

```
const sq = `SELECT *
  WHERE {
    ?s <http://xmlns.com/foaf/0.1/givenname/> ?o
  } LIMIT 10`;
context.getSparql(sq, {database: "ml-demo-app-content"},
  {headers: {'X-Example-Header': 'my-header-value'}} ).then(response => {
  console.log(response);
})
```

postSparql(combinedQuery, parameters, config)

The `postSparql(combinedQuery)` method executes a call to POST /v1/graphs/sparql. The next action depends on the results of the call:

- When the call is successful, a Promise is returned, and the `sparqlResponse` state variable is populated with the response.
- If the call is unsuccessful, an error is written to the console.

Argument	Type	Description
combinedQuery	object	A combined query object that includes a `sparql` property.
parameters	object	Object of key/value pairs corresponding to URL parameters. For details, see POST /v1/graphs/sparql.

Argument	Type	Description
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

postSparql (combinedQuery, parameters) example

```
const sq = `SELECT *
  WHERE {
    ?s <http://xmlns.com/foaf/0.1/givenname/> ?o
  } LIMIT 10`;
const combinedQuery = {
  qtext: "fast OR track",
  sparql: sq
}
context.postSparql(combinedQuery, {database: "my-app-db"}).then(response => {
  console.log(response);
})
```

postChat(message, combinedQuery, parameters, config)

The `postChat()` method executes a call to an endpoint that responds to a chat message:

- If the call is successful, a Promise returns, and the `chatResponse` state variable populates with the response.
- If the call is unsuccessful, an error is written to the console.

Argument	Type	Description
message	string	Message to submit to the backend. For example, a text prompt from the ChatBot widget.
combinedQuery	object	Optional combined query object to constrain the RAG query.
parameters	object	Object of key/value pairs that determine how to connect to the backend. Available keys are: `scheme`, `host`, `port`, `path`, `auth`. If parameter values are not defined, the values from MarkLogicProvider are used. The default `path` is `/api/fasttrack/chat`.

Argument	Type	Description
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

postChat(message, combinedQuery, parameters) example

```
const myBot = {
    id: 0,
  name: 'ExampleBot'
}
const myCombinedQuery = {
  qtext: 'suv OR crossover'
}
const myParameters={{
  host: '127.0.0.1',
  port: 4015
}}
context.postChat('What are the safest new cars?', myCombinedQuery,
myParameters)
  .then((response) => {
    console.log("Response", response);
  }
);
```

request(config)

This method executes an HTTP request based on an Axios request configuration object. The call returns a Promise. The Promise is fulfilled with an HTTP response when it is successful. The response state variable is also populated with the response.

The request uses the `scheme`, `host`, and `port` values defined by MarkLogicProvider unless these values are overridden by a baseURL property in the configuration object.

Argument	Type	Description
config	AxiosRequestConfig	A configuration object defining the HTTP request.

request(axiosRequestConfig) example

```
context.request({
      url: '/v1/resources/customEndpoint',
      method: 'POST',
      data: { prop1: 'foo', prop2: 'bar' },
      headers: { 'Content-Type': 'application/json' }
}).then(response => {
  console.log(response);
})
```

addStringFacetConstraint(constraint)

This method adds a constraint from the StringFacet widget to the context state. When a check box selection is made, this method receives the constraint object passed to the `StringFacetonSelect` event handler. Internally, this method sets a range-constraint-query.

Argument	Type	Description
constraint	object	A string facet constraint object.
constraint.type	string	The facet type ("string").
constraint.name	string	The facet name.
constraint.value	string[]	An array of selected values for the facet.
constraint.operator	string	Optional constraint operator.
constraint.title	string	The constraint title.

addStringFacetConstraint(constraint) example

```
const handleFacetClick = (constraint) => {
  context.addStringFacetConstraint(constraint);
}

return (
  <StringFacet
    title="Status"
```

```
    name="status"
    data={context.searchResponse?.facets?.status}
    onSelect={handleFacetClick}
    reset={resetStringFacet}
  />
)
```

removeStringFacetConstraint(constraint)

This method removes a string facet constraint from the context state. The method receives the constraint object passed to the SelectedFacets `removeStringFacet` event handler when a selected facet is removed. Internally, this method removes a range-constraint-query.

Argument	Type	Description
constraint	object	A string facet constraint object.
constraint.type	string	The facet type ("string").
constraint.name	string	The facet name.
constraint.value	string[]	An array of selected values for the facet.
constraint.operator	string	
constraint.title	string	The constraint title.

removeStringFacetConstraint(constraint) example

```
return (
  <SelectedFacets
    selectedFacetConfig={{
      string: {
        color: 'red',
        closable: true,
      }
    }}
    stringFacets={context.stringFacetConstraints}
```

```
    removeStringFacet={context.removeStringFacetConstraint}
  ></SelectedFacets>
)
```

addRangeFacetConstraint(constraint)

The `addRangeFacetConstraint(constraint)` method adds a range facet constraint to the context state. The method receives the constraint object passed to the DateRangeFacet `onSelect` event handler or the NumberRangeFacet and BucketRangeFacet `onChange` event handlers. Internally, this method sets a range-constraint-query.

Argument	Type	Description
constraint	object	A string facet constraint object.
constraint.type	string	The facet type. The facet type is: • "date" for DateRangeFacet constraints. • "number" for NumberRangeFacet constraints. • "string" for BucketRangeFacet constraints.
constraint.name	string	The facet name.
constraint.value	string[]	An array of selected values for the facet.
constraint.operator	string	Optional constraint operator.
constraint.title	string	The constraint title.

addRangeFacetConstraint(constraint) example

```
const handleValueRange = (constraint) => {
  context.addRangeFacetConstraint(constraint);
}

return (
  <NumberRangeFacet
```

```
    title="Salary"
    name="salary"
    data={context?.searchResponse?.facets?.salary}
    minValue={0}
    maxValue={100000}
    onChange={handleValueRange}
  />
)
```

removeRangeFacetConstraint(constraint)

The `removeRangeFacetConstraint(constraint)` method removes a range facet constraint from the context state. The method receives the constraint object passed to the SelectedFacets `removeRangeFacet` event handler when a selected facet is removed. Internally, this method removes a range-constraint-query.

Argument	Type	Description
constraint	object	A range facet constraint object.
constraint.type	string	The facet type, which is: • "date" for DateRangeFacet constraints. • "number" for NumberRangeFacet constraints. • "string" for BucketRangeFacet constraints.
constraint.name	string	The facet name.
constraint.value	string[]	An array of selected values for the facet.
constraint.operator	string	Optional constraint operator.
constraint.title	string	The constraint title.

removeRangeFacetConstraint(constraint) example

```
return (
  <SelectedFacets
```

```
    selectedFacetConfig={{
      string: {
        color: 'red',
        closable: true,
      }
    }}
    rangeFacets={context.rangeFacetConstraints}
    removeRangeFacet={context.removeRangeFacetConstraint}
  ></SelectedFacets>
)
```

addGeoConstraint(geoConstraint)

This method adds a geospatial constraint to the context state. The method receives a constraint object constructed from the coordinate array passed into the GeoMap `onGeoSearch` event handler. For more information, see addGeoConstraint(geoConstraint) example.

Argument	Type	Description
geoConstraint	object	A geospatial structured query constraint object. Object properties depend on the type of index settings and the query options defined for documents.

addGeoConstraint(geoConstraint) example

```
const handleGeoSearch = (polygonsCoordsArr) => {
  const query =
  {
    "type": "geo-elem-pair-query",
    "lat": "Latitude",
    "lon": "Longitude",
    "parent": "Location",
    "polygon": polygonsCoordsArr.map((coords) => {
      const pointObj = {
        "point": coords[0]?.map((ele) => {
          return {
            "latitude": parseFloat(ele[1]),
            "longitude": parseFloat(ele[0])
          }
        })
      }
      return pointObj
    })
```

```
  }
  context.addGeoConstraint(query);
}

return (
  <GeoMap
    markers={context.searchResponse?.results}
    sketch={true}
    transformMarkers={transformMarkers}
    transformShapes={transformShapes}
    viewType="2D"
    useWeb={false}
    showGallery={false}
    showToggle={false}
    onClickMap={onClickMap}
    onGeoSearch={handleGeoSearch}
  /> )
```

addWordQueryConstraint(constraint)

This method adds a word query constraint to the context state.

Argument	Type	Description
constraint	object	A word query constraint object

addWordQueryConstraint(constraint) example

```
context.addWordQueryConstraint(
  {
    'json-property': 'transcript',
     text: ['dog', 'cat']
  }
)
```

removeWordQueryConstraint(constraint)

This method removes a word query constraint from the context state.

Argument	Type	Description
constraint	object	A word query constraint object

removeWordQueryConstraint(constraint) example

```
context.removeWordQueryConstraint(
  {
    'json-property': 'transcript',
     text: ['dog', 'cat']
  }
)
```

removeAllWordQueryConstraints()

This method removes all existing word query constraints from the context state.

removeAllWordQueryConstraints() example

```
context.removeAllWordQueryConstraints()
```

patchComment(uri, comment)

The `patchComment(uri, comment)` method adds a comment to a document specified by a URI. This method works with the CommentBox `onSubmit` event handler. The event handler receives an object representing a comment as an argument.

Argument	Type	Description
uri	string	URI of the document for the comment.
comment	object	Comment configuration object.
comment.content	object	Comment content object.

Argument	Type	Description
comment.content.comment	string	Comment text.
comment.content.username	string	User name of the commenter.
comment.content.ts	string	Timestamp of the comment.
comment.content.imgSrc	string	Image URL associated with the comment. For example, this could point to an image of the commenter.
comment.context	string	An XPath expression that selects the node or property where the comment is inserted.
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

patchComment(uri, comment) example

```
const onSubmitComment = async (comment) => {
  let res = await context.patchComment("/person/1001.json", { content:
comment, context: '/person/comments' })
  if (res.success === true) {
    context.getDocument(selectedUri);
  } else {
    alert('Comment submission failed')
  }
}

<CommentBox
  label="Comments"
  inputPlaceholder="Add a comment"
  buttonLabel="Post"
  onSubmit={(comment) => onSubmitComment(comment)}
  username="joe-user"
  imgSrc="https://example.org/joe.jpg"
  profileImage={{
    alt: 'Avatar',
```

```
    path: 'https://example.org/joe.jpg'
  }}
/>
```

editComment(uri, id, comment)

This method edits a comment with a given ID in a document specified by a URI. The method works with the CommentList `onSaveComment` event handler. The event handler receives the comment ID and an object representing the edited comment as arguments.

Argument	Type	Description
uri	string	URI of the document for the comment.
id	string	ID of the comment.
comment	object	Comment configuration object.
comment.content	object	Comment content object.
comment.content.comment	string	Comment text.
comment.content.username	string	User name of the commenter.
comment.content.ts	string	Timestamp of the comment.
comment.content.imgSrc	string	Image URL associated with the comment. For example, the URL of an image of the user.
comment.context	string	An XPath expression that selects the node or property where the comment is edited.

Argument	Type	Description
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

editComment(uri, id, config) example

```
const onEditComment = async (id, comment) => {
  let res = await context.editComment("/person/1001.json", id, { content:
comment, context: '/person/comments/comment' })
  if (res.success !== true) {
    alert('Comment failed to be edited')
  }
}

<CommentList
  data={context.documentResponse}
  config={{comments: {
    path: 'data.person.comments.comment'
  }}}
  username="joe-user"
  onSaveComment={(id, comment) => onEditComment(id, comment)}
  onDeleteComment={(id) => onDeleteComment(id)}
```

deleteComment(uri, id, context, config)

This method deletes a comment with a given ID in a document specified by a URI. Works with the CommentList `onDeleteComment` event handler. The event handler receives the ID of the comment to delete as an argument.

Argument	Type	Description
uri	string	URI of the document with the comment to delete.
id	string	ID of the comment to delete.
context	string	An XPath expression that selects the comment node or property to delete.

Argument	Type	Description
config	AxiosRequestConfig	An optional configuration object defining the HTTP request.

deleteComment(uri, id, context) example

```
const onDeleteComment = async (id) => {
  let res = await context.deleteComment("/person/1001.json", id, '/person/
comments/comment')
  if (res.success !== true) {
    alert('Comment failed to be deleted')
  }
}

<CommentList
  data={context.documentResponse}
  config={{comments: {
    path: 'data.person.comments.comment'
  }}}
  username="joe-user"
  onSaveComment={(id, comment) => onEditComment(id, comment)}
  onDeleteComment={(id) => onDeleteComment(id)}
/>
```

MarkLogicContext state variables

This section contains information on the MarkLogicContext state variables.

chatResponse

This variable contains the result of the `postChat()` method.

chatResponse example

```
useEffect(() => {
  context.postChat(
    {
        "what is marklogic",
        {}
    })
}, []);
// chatResponse
{
```

```
    "citations": [
        {
            "citationId": "/chunks/Inside MarkLogic Server.pdf-5.json",
            "citationLabel": "[1] - Inside MarkLogic Server.pdf, Page 5"
        },
        {
            "citationId": "/chunks/Inside MarkLogic Server.pdf-5.json",
            "citationLabel": "[2] - Inside MarkLogic Server.pdf, Page 5"
        },
        {
            "citationId": "/chunks/Inside MarkLogic Server.pdf-5.json",
            "citationLabel": "[3] - Inside MarkLogic Server.pdf, Page 5"
        }
    ],
    "output": " MarkLogic Server is an Enterprise NoSQL Database that
combines database
                internals, search-style indexing, and application server
behaviors into
                a unified system.[1] It is a multi-model database management
system that
                combines a document model with a semantic triple model and
stores all of
                its data within a fully ACID-compliant transactional
repository.
                [1] MarkLogic supports massive scale and exceptional
performance while maintaining
                enterprise capabilities required for mission-critical
applications.[1]"
}
```

searchResponse

This variable contains the result of the `postSearch()` method.

searchResponse example

```
useEffect(() => {
  context.postSearch({qtext: "Alabama OR Texas"});
}, []);

// searchResponse

{
    "snippet-format": "snippet",
    "total": 2,
    "start": 1,
    "page-length": 10,
    "results": [
        {
```

```
            "index": 1,
            "uri": "/person/1002.json",
            "path": "fn:doc(\"/person/1002.json\")",
            "score": 10240,
            "confidence": 0.5028362,
            "fitness": 0.5028362,
            "href": "/v1/documents?uri=%2Fperson%2F1002.json",
            "mimetype": "application/json",
            "format": "json",
            "matches": [
                {
                    "path": "fn:doc(\"/person/1002.json\")/envelope/address/
text(\"state\")",
                    "match-text": [
                        {
                            "highlight": "Texas"
                        }
                    ]
                }
            ]
        },
        {
            "index": 2,
            "uri": "/person/1003.json",
            "path": "fn:doc(\"/person/1003.json\")",
            "score": 10240,
            "confidence": 0.5028362,
            "fitness": 0.5028362,
            "href": "/v1/documents?uri=%2Fperson%2F1003.json",
            "mimetype": "application/json",
            "format": "json",
            "matches": [
                {
                    "path": "fn:doc(\"/person/1003.json\")/envelope/address/
text(\"state\")",
                    "match-text": [
                        {
                            "highlight": "Alabama"
                        }
                    ]
                }
            ]
        }
    ],
    "qtext": "Alabama OR Texas",
    "metrics": {
        "query-resolution-time": "PT0.005024S",
        "snippet-resolution-time": "PT0.000615S",
        "total-time": "PT0.315651S"
    }
}
```

searchResponseLoading

This variable is `true` when the asynchronous `postSearch()` method has been executed and is waiting for a response and `false` otherwise. The variable can be checked to display a loading message during a search.

searchResponseLoading example

```
{ context.searchResponseLoading ?
  <div>Loading...</div> :
  <ResultsSnippet
    results={context.searchResponse.results}
  />
}
```

suggestedSearchResponse

This variable is the result of the `getSuggestedSearch()` method.

suggestedSearchResponse example

```
useEffect(() => {
  context.getSuggestedSearch("New", 5)
}, []);

// suggestedSearchResponse

{
    "suggestions": [
        "\"New Haven\"",
        "\"New York City\"",
        "Newark",
        "Newton"
    ]
}
```

documentResponse

This variable is the result of the `getDocument()` method. The object's `data` property is set to the document content and the object's `uri` property is set to the document URI.

documentResponse example

```
useEffect(() => {
  context.getDocument({"/person/1001.json", {database: "my-app-db"})
}, []);
```

```
// documentResponse

{
  data: {
    id: "1001",
    first: "Jane",
    last: "Smith"
  },
  uri: "/person/1001.json"
}
```

sparqlResponse

Result of the `getSparql()` or `postSparql()` methods.

sparqlResponse example

```
useEffect(() => {
  context.getSparql(`
    SELECT *
    WHERE {
    ?s ?p ?o .
    ?s <http://xmlns.com/foaf/0.1/knows/> ?o
    } LIMIT 10
  `);
}, []);

// sparqlResponse

{
    "head": {
        "vars": [
            "s",
            "p",
            "o"
        ]
    },
    "results": {
        "bindings": [
            {
                "s": {
                    "type": "uri",
                    "value": "http://example.org/1001"
                },
                "p": {
                    "type": "uri",
                    "value": "http://xmlns.com/foaf/0.1/knows/"
                },
```

```
                "o": {
                    "type": "literal",
                    "value": "1002"
                }
            },
            {
                "s": {
                    "type": "uri",
                    "value": "http://example.org/1001"
                },
                "p": {
                    "type": "uri",
                    "value": "http://xmlns.com/foaf/0.1/knows/"
                },
                "o": {
                    "type": "literal",
                    "value": "1003"
                }
            },
            {
                "s": {
                    "type": "uri",
                    "value": "http://example.org/1002"
                },
                "p": {
                    "type": "uri",
                    "value": "http://xmlns.com/foaf/0.1/knows/"
                },
                "o": {
                    "type": "literal",
                    "value": "1001"
                }
            },
            {
                "s": {
                    "type": "uri",
                    "value": "http://example.org/1003"
                },
                "p": {
                    "type": "uri",
                    "value": "http://xmlns.com/foaf/0.1/knows/"
                },
                "o": {
                    "type": "literal",
                    "value": "1001"
                }
            }
        ]
    }
}
```

response

This variable is the result of the `request()` method.

response example

```
useEffect(() => {
  context.request({
    url: '/v1/resources/currentTime',
    method: 'GET',
    headers: { 'Content-Type': 'application/json' }
  });
}, []);

// response

{
  currentTime:"10:22:25-07:00"
}
```

stringFacetConstraints

Current string facet constraints of the context state. Constraints may be added with `addStringFacetConstraint()` and removed with `removeStringFacetConstraint()`.

stringFacetConstraints example

```
useEffect(() => {
  context.addStringFacetConstraint([
    {
        "type": "string",
        "name": "status",
        "value": [
            "Active"
        ],
        "title": "Status"
    }
])
}, []);

// stringFacetConstraints

[
    {
        "type": "string",
        "name": "status",
        "value": [
            "Active"
```

```
        ],
        "title": "Status"
    }
]
```

rangeFacetConstraints

The `rangeFacetConstraints` state variable holds the current range facet constraints of the context state. Constraints may be added with `addRangeFacetConstraint()` and removed with `removeRangeFacetConstraint()`.

rangeFacetConstraints example

```
useEffect(() => {
  context.addRangeFacetConstraint([
    [
        {
            "type": "number",
            "name": "salary",
            "value": 50000,
            "operator": "GE",
            "title": "Salary"
        },
        {
            "type": "number",
            "name": "salary",
            "value": 100000,
            "operator": "LE",
            "title": "Salary"
        }
    ]
  ])
}, []);

// rangeFacetConstraints

[
    [
        {
            "type": "number",
            "name": "salary",
            "value": 50000,
            "operator": "GE",
            "title": "Salary"
        },
        {
            "type": "number",
```

```
            "name": "salary",
            "value": 100000,
            "operator": "LE",
            "title": "Salary"
        }
    ]
]
```

wordQueryConstraints

The `wordQueryConstraints` state variable holds the current word query constraints of the context state. A constraint may be added with `addWordQueryConstraint()`. A constraint may be removed with `removeWordQueryConstraint()`, or all constraints may be removed at once with `removeAllWordQueryConstraints()`.

wordQueryConstraints example

```
   useEffect(() => {
        context.addWordQueryConstraint(
          {
              "json-property": "transcript",
              "text": ['dog', 'cat']
          })
        }, []);
// wordQueryConstraint
    {
        "json-property": "transcript",
        "text": ['dog', 'cat']
    }
```

NetworkGraph

The NetworkGraph widget displays relationships between entities. The relationships are displayed as nodes connected by lines. A configuration allows users to filter the nodes displayed in the graph.

NetworkGraph MarkLogic setup

The NetworkGraph widget displays graphs showing relationships from MarkLogic result sets. For example, the result sets from the `/v1/search` and `/v1/graphs/sparql` endpoints from the MarkLogic REST API can be graphed using the widget.

In order to display triples from `/v1/search` results, the content must be extracted and included in the search results. This can be done by using the `extract-document-data` property in the query options in

the application. For more information, see Include document extracts in search results.

> **Note:**
>
> To use the `/v1/graphs/sparql` endpoint to execute SPARQL queries on triples and display the SPARQL results as a graph, the database triples index must be turned on.

Graph data model

To configure the NetworkGraph widget, an object with key/value pairs is provided that represents nodes and links in the graph. Each node and link have a unique key with a value object that determines how they appear in the graph. Link objects have `id1` and `id2` properties that use node keys as their values. The node keys determine the source and destination of each link.

In this example, the NetworkGraph widget configuration object has three nodes and three links. The configuration object is passed into `NetworkGraph` using the `items` prop. Additional configuration values are passed into the widget using the `settings` prop. The `options` object turns on navigation buttons (`navigation: true`) and sets the canvas background to white `(backgroundColor: 'white')`.

For more information, see the third-party documentation included with FastTrack.

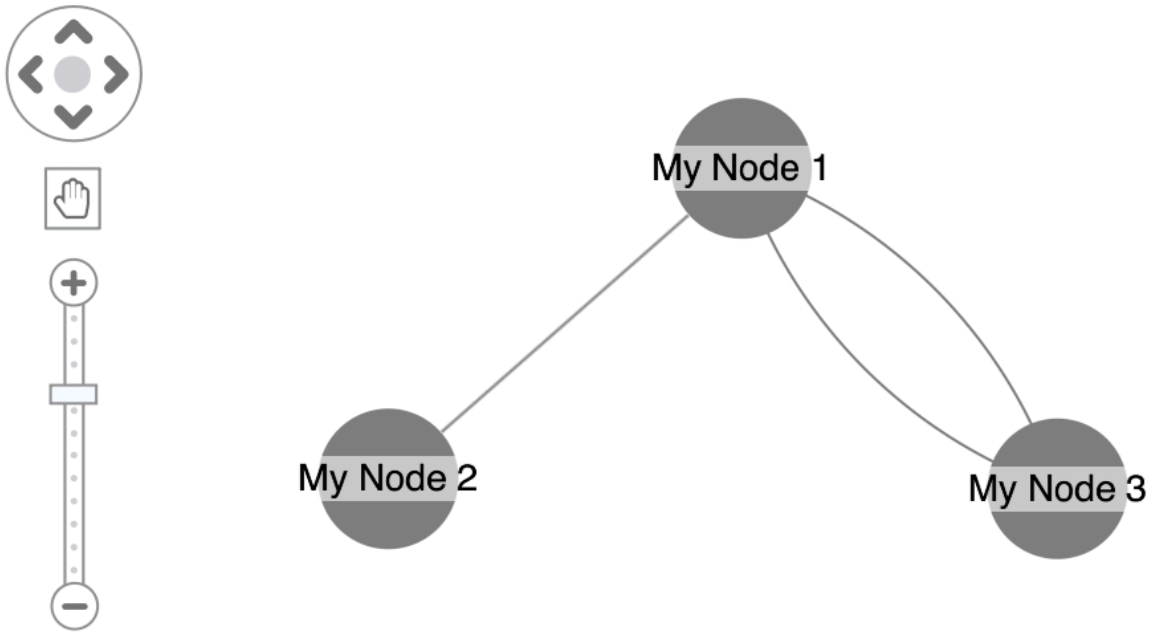
```
import { NetworkGraph } from "ml-fasttrack";

function App() {

  const items = {
    node1: {
      label: [{ text: 'My Node 1' }]
    },
    node2: {
      label: [{ text: 'My Node 2' }]
    },
    node3: {
      label: [{ text: 'My Node 3' }]
    },
    link1: {
      id1: 'node1',
      id2: 'node2'
    },
    link2: {
      id1: 'node1',
      id2: 'node3'
    },
    link3: {
      id1: 'node3',
      id2: 'node1',
```

```
    }
  };

  const settings = {
    options: {
      navigation: true,
      backgroundColor: 'white',
    }
  };

  return (
    <div>
      <NetworkGraph
        items={items}
        settings={{
          options: {
            backgroundColor: "white",
            navigation: true,
          }
        }}
      />;
    </div>
  );
}

export default App;
```

> **Note:**
>
> The example displays nodes and graphs in NetworkGraph by defining them directly in an `items` object. The example does not transform data from MarkLogic into an object.
>
> The next sections describe how to transform MarkLogic data and display it in a graph. The configuration displays the NetworkGraph example rendering.

NetworkGraph example rendering

In this example, one link goes from **My Node 1** to **My Node 2**, another goes from **My Node 1** to **My Node 3**, and a third goes from **My Node 3** to **My Node 1**. Labels are added to the nodes as part of the node configuration.



There are many other configuration options for the nodes, links, and graph canvas.

For more information, see the third-party documentation included with FastTrack.

Graph search results with entityConfig

Relationship data from embedded triples in `/v1/search` results can be displayed as a graph. This is accomplished using the `data` and `entityConfig` props. The `data` prop is set to the `/v1/search` response. The `entityConfig` prop maps responses to the graph. This strategy assumes that:

- Documents in the search results represent entity instances that will be represented as nodes in the graph. Node URIs are used as node keys.
- Document URIs can be derived from the relationship triples embedded in the documents. If the data is modeled differently, refer to Graph results with transforms to transform relationship data.

In this example, the subject, predicate, and object are graphed from each embedded triple in a `/v1/search` response. The search response looks like this:

```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
    "results": [
        {
            "index": 1,
```

```
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "address": {
                                "state": "Texas",
                                "country": "United States"
                            },
                            "relations": [
                                {
                                    "triple": {
                                        "subject": "http://example.org/
person/1001.json",
                                        "predicate": "http://xmlns.com/foaf/
0.1/knows/",
                                        "object": "http://example.org/person/
1002.json"
                                    }
                                },
                                {
                                    "triple": {
                                        "subject": "http://example.org/
person/1001.json",
                                        "predicate": "http://xmlns.com/foaf/
0.1/knows/",
                                        "object": "http://example.org/person/
1003.json"
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        // more results
    ],
    "qtext": "person",
}
```

Each search result includes a document extract with semantic triples. The `subject` and `object` values in the triples include the URIs of the entities they relate to. Each search result also includes the `entityType` for the entity instance. In this example, `entityType` is set to `person`.

The NetworkGraph widget can be configured to display entity instances as nodes in the graph. Links

represent the triple relationships.

NetworkGraph example configuration

A React application with the NetworkGraph widget and `entityConfig` and `data` props looks like this:

```
import { useContext } from "react";
import "./App.css";
import { MarkLogicContext, SearchBox, NetworkGraph } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const entityConfig = {
    entityTypeConfig: {
      "path": "extracted.content[0].envelope.entityType"
    },
    entities: [
      {
        entityType: "person",
        triples: {
          path: "extracted.content[0].envelope.relations",
          subject: {
            path: "triple.subject",
            regex: /\/\w+\/[^/]+$/
          },
          predicate: {
            path: "triple.predicate"
          },
          object: {
            path: "triple.object",
            regex: /\/\w+\/[^/]+$/
          }
        },
        items: {
          label: [
            {
              text: { path: "extracted.content[0].envelope.firstName" }
            }
          ]
        },
        nodeRelations: {
          relationLabelRgx: /(\w+)$/
        }
      }
    ]
  }
```

```
  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div>
          <NetworkGraph
            data={context?.searchResponse}
            entityConfig={entityConfig}
          />
      </div>
    </div>
  );
}

export default App;
```

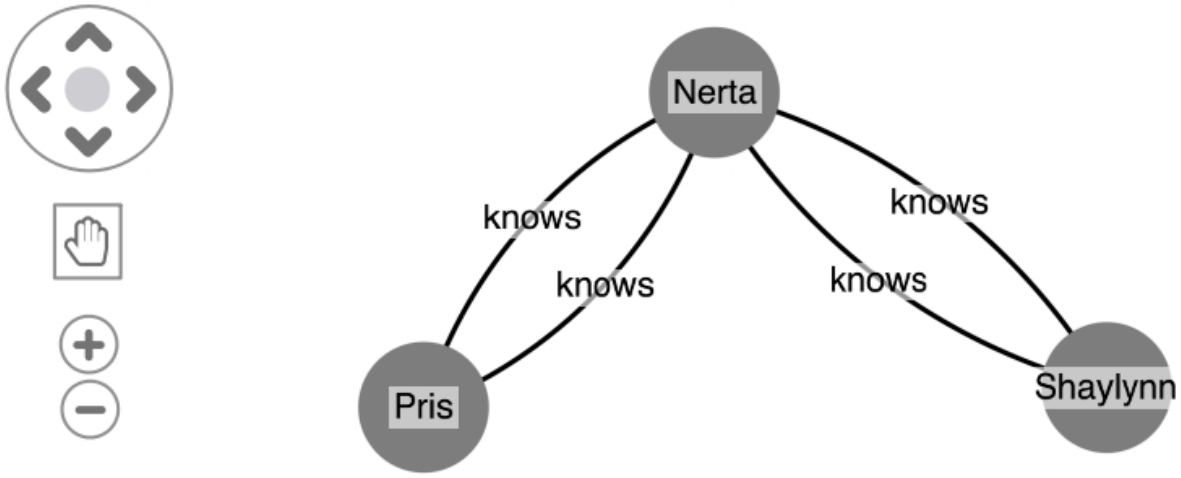
Code explanation

Notes on the NetworkGraph example configuration:

- To make the MarkLogic search response available to the NetworkGraph widget, the `data` prop is assigned the `searchResponse` value from the application context.
- The `entityConfig` prop accepts a configuration object that tells the widget how to build the data model for the underlying graph from the response data.
- The path values in the configuration object map the data in the search response payload to the graph data model.
- Regex props can be used to select parts of the triple values to use in the graph. This example uses the regex values to extract the URIs from the subject and object values. For more details about the configuration props, see NetworkGraph API

NetworkGraph example rendering

The NetworkGraph example configuration generates this graph:



Node labels are defined in the `firstName` property from each entity instance and are configured with the `entities[].items property`. The link label is extracted from the `predicate` value in each triple using the regex in the `nodeRelations` property.

Graph results with transforms

Data from SPARQL results and other types of payloads can be transformed into the data model expected by the graph widget. Then, the data can be passed to the NetworkGraph widget as an `items` prop.

This example shows a React application using a SPARQL query and the transformation strategy:

```
import { useContext, useEffect } from "react";
import "./App.css";
import { MarkLogicContext, NetworkGraph } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const sparqlQuery = `
    SELECT (str(?subName) as ?s) ?p (str(?objName) as ?o) (str(?subState) as
?sState) (str(?objState) as ?oState)
    WHERE {
      ?subject ?predicate ?object .
      ?subject <http://xmlns.com/foaf/0.1/knows> ?object .
      ?subject <http://example.org/firstName> ?subName .
      ?object <http://example.org/firstName> ?objName .
      ?subject <http://example.org/state> ?subState .
      ?object <http://example.org/state> ?objState .
      BIND (SUBSTR(?predicate,27,5) AS ?p)
```

```
    }
  `

  const sparqlToItems = (sparqlResponse) => {
    if (!sparqlResponse) return;
    let items = {};
    sparqlResponse.results.bindings.forEach(r => {
      const { s, p, o } = r
      // create subject node
      if (s && s?.value) {
          items[s.value] = {
            label: [{ text: s?.value }],
          }
      }
      // create object node
      if (o && o?.value) {
          items[o.value] = {
            label: [{ text: o?.value }],
          }
      }
      // create predicate link
      if (s && s?.value && o && o?.value) {
        items[s.value + '-' + o.value] = {
          id1: s.value,
          id2: o.value,
          label: { text: p?.value },
        }
      }
    })
    return items;
  }

  useEffect(() => {
    context.getSparql(sparqlQuery);
  }, []);

  return (
    <div className="App">
      <div>
          <NetworkGraph
            items={sparqlToItems(context.sparqlResponse)}
            onSelectNode={(event) => console.log(event)}
          />
      </div>
    </div>
  );
}

export default App;
```

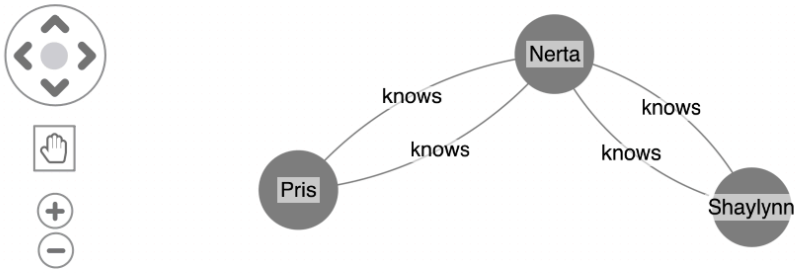
Code explanation

In Graph results with transforms:

- The SPARQL query is used on the `/v1/graphs/sparql` endpoint using the `getSparql` method from the application context. The query is defined in the `sparqlQuery` variable.
- The query is executed on load using a `useEffect` hook and the `getSparql` method from the application context.
- The SPARQL result is stored in `context.sparqlResponse` by the application context.
- To transform the MarkLogic SPARQL response into the data model expected by the graph widget, a `sparqlToItems` function is used. It is a custom function that accesses the bindings in a SPARQL response to build nodes and links.
- The `onSelectNode` prop is passed a callback to handle node selection events.

Graph results with transforms example rendering

The code in Graph results with transforms generates this graph. Clicking a node in the graph logs the click event to the console.



Styling nodes and links

Styles can be applied to nodes and links in the graph by adding properties. The examples in this section illustrate how to:

- change the color and size of nodes.
- change the color and width of links.
- turn on link arrows.

If the entityConfig strategy is used for mapping, additional properties can be added to the `entityConfig` object under the `entities[].items` and `entities[].nodeRelations` properties:

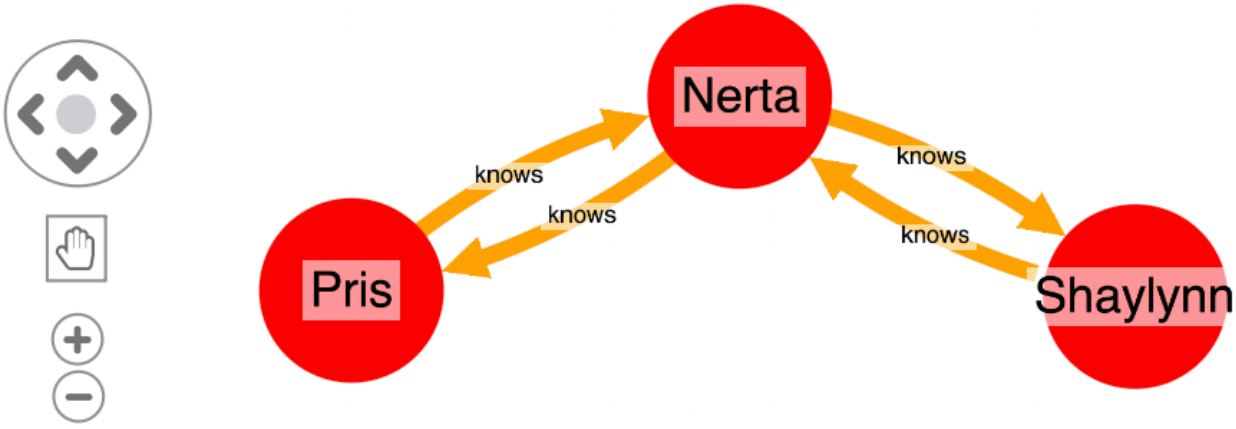
```
const entityConfig = {
    entityTypeConfig: {
      "path": "extracted.content[0].envelope.entityType"
    },
    entities: [
      {
        entityType: "person",
        triples: {
          path: "extracted.content[0].envelope.relations",
          subject: {
            path: "triple.subject",
            regex: /\/\w+\/[^/]+$/
          },
          predicate: {
            path: "triple.predicate"
          },
          object: {
            path: "triple.object",
            regex: /\/\w+\/[^/]+$/
          }
        },
        items: {
          color: "red",
          size: 2,
          label: [
            {
              text: { path: "extracted.content[0].envelope.firstName" }
            }
          ]
        },
        nodeRelations: {
          relationLabelRgx: /(\w+)$/,
          link: {
            width: 10,
            color: "orange",
            end1: { arrow: false },
            end2: { arrow: true }
          }
        }
      }
    ]
  }
```

If a transformation function is used for mapping, the style properties can be set for each subject, predicate, and object in the function:

```
const sparqlToItems = (sparqlResponse) => {
  if (!sparqlResponse) return;
  let items = {};
  sparqlResponse.results.bindings.forEach(r => {
    const { s, p, o } = r
```

```
    if (s && s?.value) {
        items[s.value] = {
          label: [{ text: s?.value }],
          color: "red",
          size: 2
        }
    }

    if (o && o?.value) {
        items[o.value] = {
          label: [{ text: o?.value }],
          color: "red",
          size: 2
        }
    }

    if (s && s?.value && o && o?.value) {
      items[s.value + '-' + o.value] = {
        id1: s.value,
        id2: o.value,
        label: { text: p?.value },
        width: 3,
        color: "orange",
        end1: { arrow: false },
        end2: { arrow: true }
      }
    }

  })
  return items;
}
```

Styling the nodes and links using the additional properties displays this graph:



For more information, see the third-party documentation included with FastTrack.

Filter nodes

Nodes in the graph can be filtered. To filter nodes, include properties for the filter in each node's configuration. If the entityConfig strategy is used to show search results as a graph, include a `filters` object with key/value pairs. The key/values pairs should specify the path to the result property. This example includes the state address in each node configuration with the key `stateVal`:

```
const entityConfig = {
    entityTypeConfig: {
      "path": "extracted.content[0].envelope.entityType"
    },
    entities: [
      {
        entityType: "person",
        triples: {
          path: "extracted.content[0].envelope.relations",
          subject: {
            path: "triple.subject",
            regex: /\/\w+\/[^/]+$/
          },
          predicate: {
            path: "triple.predicate"
          },
          object: {
            path: "triple.object.value",
            regex: /\/\w+\/[^/]+$/
          }
        },
        filters: {
          stateVal: { path: 'extracted.content[0].envelope.address.state' },
        },
        nodeRelations: {
          relationLabelRgx: /(\w+)$/
        }
      }
    ]
  }
```

Custom transformation functions can include filter properties in each node configuration. In this example SPARQL query, state values for the subjects and objects are included as the `sState` and `oState` bindings. These bindings can then be accessed in order to put the state values in the subject and object node configurations:

```
const sparqlToItems = (sparqlResponse) => {
    if (!sparqlResponse) return;
    let items = {};
    sparqlResponse.results.bindings.forEach(r => {
      const { s, p, o, sState, oState } = r
```

```
      if (s && s?.value) {
          items[s.value] = {
            label: [{ text: s?.value }],
            stateVal: sState.value
          }
      }

      if (o && o?.value) {
          items[o.value] = {
            label: [{ text: o?.value }],
            stateVal: oState.value
          }
      }

      if (s && s?.value && o && o?.value) {
        items[s.value + '-' + o.value] = {
          id1: s.value,
          id2: o.value,
          label: { text: p?.value },
        }
      }

    })
    return items;
  }
```

Once the filter values are present in the node configuration objects, a filter list can be configured in the NetworkGraph widget with a `filterConfig` prop:

```
<NetworkGraph
  data={context?.searchResponse}
  entityConfig={graphConfig}
  filterConfig={[
    {
      label: 'State',
      path: 'stateVal'
    }
  ]}
/>
```

Code explanation

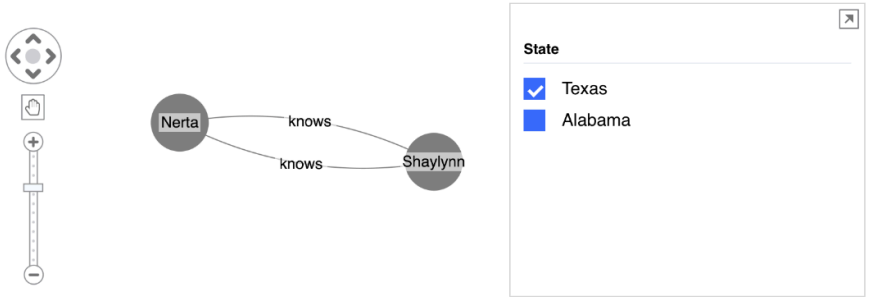
In the Filter nodes example:

- Each object in the `filterConfig` array defines a filter list. This example defines a single filter list for the `state` property.
- The `label` value specifies the list title.
- The `path` value references the property in the node object. This path value can be the key from the filter's

configuration object (in this case, `stateVal`) or a dot-notated path from the key if the value in the node object is an object.

Filter nodes example rendering

Adding a filter list to the NetworkGraph using the example code in Filter nodes displays:



Users can click the check boxes to limit the nodes displayed.

NetworkGraph API

Prop	Type	Description
items	object	Items to display in the graph. Each node and relation are defined with a property and object value. For more information, see the third-party documentation included with FastTrack.

Note:

Nodes and links can be defined

Prop	Type	Description
		in the graph directly using the `items` prop.
data	object	Search results payload to display in the graph. This works in conjunction with the `entityConfig` prop.
dataConfig	object	Optional JSONPath for indicating the results in the data prop.
entityConfig	object	Entity-specific configuration settings for the subject, predicate, and object values, labels, and styles in the graph. For more information, see the third-party documentation included with FastTrack.
showMap	boolean	Show the map in the canvas with Leaflet. Requires a coordinates object. For more information, see the third-party documentation included with FastTrack.

Prop	Type	Description
settings	Record<string, any>	Configuration props passed in for the graph, including canvas options, event handlers, and other properties (such as adding a reference to manage adjustments in the view). For more information, see the third-party documentation included with FastTrack.
height	string	Height of the widget canvas as a CSS height value. Default is `"400px"`.
width	string	Width of the widget canvas as a CSS width value. Default is `"100%"`
relationsLevel	number	Relations level for filtering.
selectedElement	string	ID of the selected element.
filterConfig	{ id: string; label: string; path: string; }[]	Configuration settings for the graph filter.
fontFamilies	string[]	Array of font families. Example name: "Font Awesome 5 Free".
nodeConfig	object	Default settings for a node when it is not defined in `entityConfig`.
relationConfig	object	Default settings for a relation link when it is not defined in entityConfig.
itemsTooltipConfig	object	Configuration object that maps and formats a tooltip for a node when not defined in `entityConfig`.

Prop	Type	Description
onSelectNode	((node: any) => void)	Callback function triggered when a node is clicked.
onDoubleClickNode	((node: any) => void)	Callback function triggered when a node is double-clicked.

entityConfig API

Property	Type	Description
entityTypeConfig	PathConfig	Entity type configuration object.
entityTypeConfig.path	string	Path to the entity type in the search result. The path is specified using JSONPath .
entities[]	object[]	Array of graph configuration objects for each entity.
entities[].entityType	string	Entity type of the configuration object.
entities[].triples	object	Triples configuration object.
entities[].triples.path	string	Path to the array of triples. The path is specified using JSONPath.
entities[].triples.subject	PathConfig	Triple subject configuration object.
entities[].triples.subject.path	string	Path to the subject value relative to the triples path. The path is specified using JSONPath.

Property	Type	Description
entities[].triples.subject.regex	regex	Optional regex with which to select a part of the subject value.
entities[].triples.predicate	PathConfig	Triple predicate configuration object.
entities[].triples.predicate.path	string	Path to the predicate value relative to the triples path. The path is specified using JSONPath.
entities[].triples.object	PathConfig	Triple object configuration object.
entities[].triples.object.path	string	Path to the object value relative to the triples path. The path is specified using JSONPath.
entities[].triples.object.regex	regex	Optional regex with which to select a part of the object value.
entities[].items	object	Styles and formatting applied to the nodes. For more information, see the third-party documentation included with FastTrack.
entities[].nodeRelations.relationLabelRgx	string	Optional regex with which to select a part of the predicate value as a link label. DEPRECATED. Use entities[].triples.predicate.regex instead.
entities[].nodeRelations.link	object	Styles and formatting applied to the links. For more information, see the third-party documentation included with FastTrack.
entities[].tooltipConfig	object	Configuration object that maps and formats a tooltip for a node.

Property	Type	Description
entities[].filters	object	Configuration object that defines extra values to include in the node objects. The values are used for filtering.

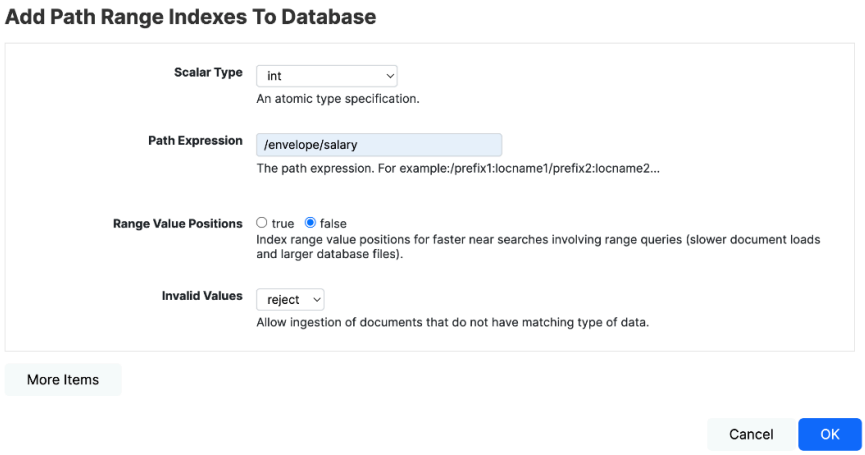
NumberRangeFacet

The NumberRangeFacet widget displays a range slider for a faceted numeric property in search results. Users can select start and end values to constrain their search.

> **Note:**
>
> Users can also constrain a search on numeric values using the BucketRangeFacet widget.

NumberRangeFacet MarkLogic setup

Faceted search in MarkLogic requires a range index on the faceted property. Add a range index for a property in the database configuration using the Admin Interface or one of MarkLogic's programmatic APIs. This screenshot shows a path range index being added to the salary property in the Admin Interface:



With an index in place, a constraint can be configured in the query options of the search application. This will return facets for the property in the search results. For example:

```
?xml version="1.0" encoding="UTF-8"?>
<options xmlns="http://marklogic.com/appservices/search">
   <constraint name="salary">
        <range type="xs:int" facet="true" collation="">
            <path-index>/envelope/salary</path-index>
        </range>
   </constraint>
   <return-facets>true</return-facets>
</options>
```

The constraint settings correspond to the settings for the index. `return-facets` is set to `true` so that facet results are returned with search results.

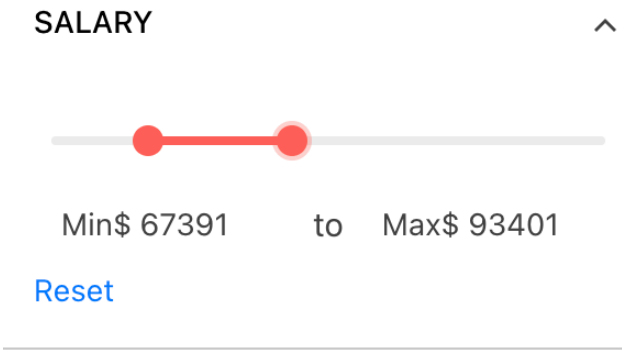
This example shows `salary` facet information returned in the search response:

```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "score": 0,
            "confidence": 0,
            "fitness": 0,
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "mimetype": "application/json",
            "format": "json",
            "matches": [
                {
                    "path": "fn:doc(\"/person/1001.json\")/object-node()",
                    "match-text": [
                        "person Nerta Hallwood Marketing Manager Active
1985-03-04 104000 person-1001.jpg"
                    ]
                }
            ],
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
```

```
                            "status": "Active",
                            "dob": "1985-03-04",
                            "salary": 104000,
                            "image": "person-1001.jpg",
                        }
                    }
                ]
            }
        },
        // ...
    ],
    "facets": {
        "salary": {
            "type": "xs:int",
            "facetValues": [
                {
                    "name": "55000",
                    "count": 1,
                    "value": 55000
                },
                {
                    "name": "87000",
                    "count": 1,
                    "value": 87000
                },
                {
                    "name": "104000",
                    "count": 1,
                    "value": 104000
                }
            ]
        }
    },
    // ...
}
```

Example rendering

This example shows the NumberRangeFacet widget. The widget displays a range slider for a faceted salary property. Only documents with a `salary` property between $67391 and $93401 are returned and displayed.



NumberRangeFacet example configuration

This configuration shows the NumberRangeFacet widget imported and configured in a React application:

```
import { useContext, useState, useEffect } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsSnippet, NumberRangeFacet } from
"ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const [numberVals, setNumberVals] = useState([])

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const updateNumberRange = (selections) => {
    setNumberVals(selections)
  }

  const resetNumberRange = () => {
    context.removeRangeFacetConstraint(numberVals)
    setNumberVals([])
  }

  useEffect(() => {
    const debounceTimeout = setTimeout(() => {
      if (numberVals.length !== 0) {
        context.addRangeFacetConstraint(numberVals)
      }
    }, 500)
    return () => {
      clearTimeout(debounceTimeout);
    };
```

```
  }, [numberVals]);

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div style={{display: 'flex', flexDirection: 'row'}}>
          <div style={{width: '640px'}}>
            <ResultsSnippet
              results={context.searchResponse.results}
              paginationFooter={true}
            />
          </div>
          <div>
            {context?.searchResponse?.facets?.salary &&
              <NumberRangeFacet
                title="Salary"
                data={context?.searchResponse?.facets?.salary}
                name="salary"
                separator={'to'}
                minLabel={'Min'}
                maxLabel={'Max'}
                prefix={"$"}
                suffix={""}
                minValue={50000}
                maxValue={150000}
                onChange={updateNumberRange}
                onReset={resetNumberRange}
              />
            }
          </div>
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In the NumberRangeFacet example configuration:

- The `data` prop is set to the numeric facet displayed from the search response object.
- The selections information from the NumberRangeFacet widget is stored in a `numberVals` state variable by the `onChange` callback. This information is cleared by the `onReset` callback.
- Using a `debounceTimeout` function avoids unneeded updates to the context object (which results in extra searches on the backend).

NumericRangeFacet API

Prop	Type	Description
data	object	Facet data to display from search results.
title	string	Title for the collapsible header.
subTitle	string	Subtitle for the collapsible header.
name	string	String identifying the facet. Passed as the name value in the onChange event.
separator	string	Separator displayed between the minimum and maximum cards.
minLabel	string	Label for the minimum (start) value.
maxLabel	string	Label for the maximum (end) value.
prefix	string	Optional string to be displayed before the numeric values.
suffix	string	Optional string to be displayed after the numeric values.
containerStyle	CSSProperties	CSS styles applied to the widget.
cardsStyle	CSSProperties	CSS styles applied to the minimum and maximum cards.

Prop	Type	Description
step	number	Step by which the number value is incremented/decremented.
minValue	number	Minimum value for the range.
maxValue	number	Maximum value for the range.
onChange	function	Callback function triggered when the range slider values change. It receives an array of selection objects, see NumberRangeFacet callbacks.
onReset	function	Callback function for the reset event.

NumberRangeFacet callbacks

This example shows the `selections` object passed to the `onChange` callbacks:

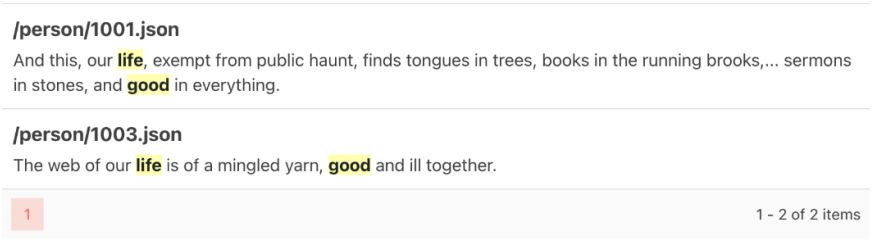
```
[
    {
        "type": "number",
        "name": "salary",
        "value": 67391,
        "operator": "GE",
        "title": "Salary"
    },
    {
        "type": "number",
        "name": "salary",
        "value": 93401,
        "operator": "LE",
        "title": "Salary"
    }
]
```

ResultsSnippet

The ResultsSnippet widget displays MarkLogic search results as document URIs and search snippets. Search snippets are portions of matching documents. The matching text in the search snippets is highlighted.

ResultsSnippet example rendering

In this example, the ResultsSnippet widget displays results for the search "life AND good". Pagination is turned on in the widget footer.



ResultsSnippet configuration

In this example, the ResultsSnippet widget displays search results returned from MarkLogic using `MarkLogicContext`. The `results` prop accepts a search results object from the context.

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsSnippet } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  return (
    <div className="App">
      <div className="SearchBox">
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div>
        <ResultsSnippet
          results={context.searchResponse.results}
          paginationFooter={true}
```

```
        />
      </div>
    </div>
  );
}

export default App;
```

ResultsSnippet API

Prop	Type	Description
results	object	Search results data to display.
highlightBg	string	HTML background color for the highlighted matching text. Default: `"#FFFFB0"`.
highlightText	string	HTML color for the highlighted matching text. Default: `"#212529"`.
highlightWeight	string	CSS font-weight value for the highlighted matching text. Default: `"bold"`.
className	string	Class name applied to the widget.
onClick	((result: any) => void)	Callback function triggered when a result item is clicked. Receives the result object.
onMouseEnter	((result: any) => void)	Callback function triggered when the mouse pointer enters a result item. Receives the result object.

Prop	Type	Description
onMouseLeave	((result: any) => void)	Callback function triggered when the mouse pointer leaves a result item. Receives the result object.
onScroll	((event: ListViewEvent) => void)	Callback function triggered when the widget has been scrolled. Receives an event object.
titleStyle	string	CSS styles applied to each result title.
snippetStyle	string	CSS styles applied to each result snippet.
resultStyle	string	CSS styles applied to each result.
headerClassName	string	Class name applied to the widget header.
headerValue	ReactNode	Content displayed in the widget header.
footerClassName	string	Class name applied to the widget footer.
footerValue	ReactNode	Content to display in the widget footer.
paginationHeader	boolean	Indicates whether to display pagination controls in the header.
paginationFooter	boolean	Indicates whether to display pagination controls in the footer.

Prop	Type	Description
pageSizeChanger	number[] | (string | number)[]	Numeric array for displaying a menu in the pagination for configuring the number of items on each page.
showPreviousNext	boolean	Indicates whether to display previous and next buttons in the pagination.
showInfoSummary	boolean	Optional value for displaying the results summary in the pagination options.
paginationClassName	string	Class name applied to the pagination container.
paginationSize	"small" | "medium" | "large"	Size of the pagination.
pagerButtonCount	number	The number of page buttons to display in the pagination controls.

ResultsConfig

The ResultsConfig widget displays a list of search results from a POST `/v1/search` response from the MarkLogic REST API. The widget maps content in each search result to a title and labeled items beneath the title.

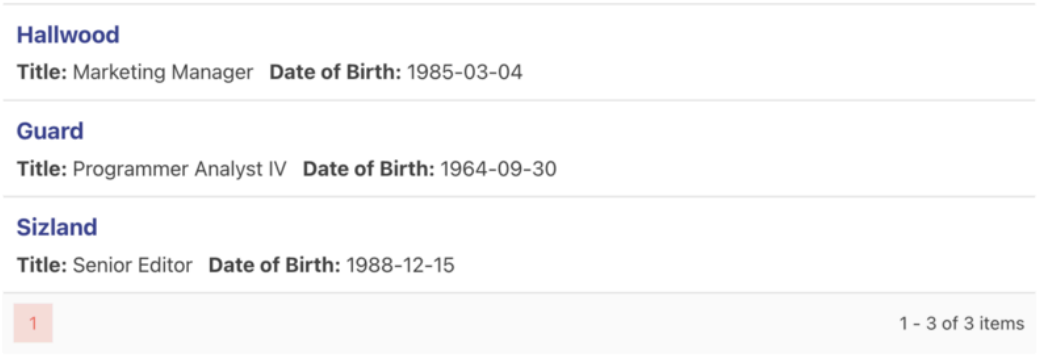
The ResultsConfig widget is similar to the ResultsCustom widget, which maps search results content using a callback function passed as a prop.

ResultsConfig MarkLogic configuration

To display content from search result documents in the ResultsConfig widget, the content must be extracted and included in search results. To do this, include the `extract-document-data` property in the query options of the application. See Include document extracts in search results.

The ResultsConfig widget expects to receive search results in the `results` prop. The expected format is the format returned from POST /v1/search in the MarkLogic REST API.

ResultsConfig example rendering



The example shows results displayed with the widget using the `config` object prop for defining how the content is mapped. For each search result, a person's last name is mapped to the title. Job title and date of birth are mapped to the labeled items beneath the title.

ResultsConfig example configuration

In this example:

- The ResultsConfig widget is passed a `config` object prop that maps content in each search result to the title and labeled items beneath the title.
- The `config` object supports different renderings for multiple entity types in the search results.
- The `entityTypeConfig` object specifies where the entity type is specified in the search results.
- The `entities` array accepts a separate `config` object for each entity type. In this example, `person` is the single entity type.

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsList } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  return (
    <div className="App">
      <div className="SearchBox">
        <SearchBox onSearch={handleSearch}/>
```

```
      </div>
      <ResultsList
        results={context.searchResponse.results}
        paginationFooter={true}
        config={{
          entityTypeConfig: {
            path: 'extracted.content[0].envelope.entityType'
          },
          entities: [
            {
              entityType: 'person',
              title: {
                path: 'extracted.content[0].envelope.lastName'
              },
              items: [
                {
                  label: 'Title',
                  path: 'extracted.content[0].envelope.title'
                },
                {
                  label: 'Date of Birth',
                  path: 'extracted.content[0].envelope.dob'
                }
              ],
            }
          ]
        }}
      />
    </div>
  );
}

export default App;
```

ResultsConfig API

Prop	Type	Description
results	object[]	Search results data to display.
config	object	Array of entity configuration objects. Each object determines what data to display in a result for an entity type.

Prop	Type	Description
config.entityTypeConfig	PathConfig	Entity type configuration object.
config.entityTypeConfig.path	object	The path to the entity type in the search result. The path is specified using JSONPath.
config.entities[]	object[]	An array of search result configuration objects.
config.entities[].entityType	string	The entity type of the configuration object.
config.entities[].title	PathConfig	A title configuration object.
config.entities[].title.path	string	The path to the title in the search result. The path is specified using JSONPath.
config.entities[].items[]	PathConfig	An array of configuration objects for the items beneath the title.
config.entities[].items[].label	string	The item label.
config.entities[].items[].path	string	The path to the item value in the search result. The path is specified using JSONPath.
title	string	Optional title for the results list.
titleStyle	CSSProperties	CSS styles applied to the results list title.
itemTitleStyle	CSSProperties	CSS styles applied to the title of each result.
labelStyle	CSSProperties	CSS styles applied to the result item labels.

Prop	Type	Description
valueStyle	CSSProperties	CSS styles applied to the result item values.
itemsContainerStyle	CSSProperties	CSS styles applied to each item container.
multipleValueSeparator	string	The string separator between array values.
className	string	Class name applied to the widget.
headerClassName	string	Class name applied to the widget header.
headerValue	any	Content displayed in the widget header.
footerClassName	number	Class name applied to the widget footer.
footerValue	any	Content to display in the widget footer.
paginationHeader	boolean	Indicates whether to display pagination controls in the header.
paginationFooter	boolean	Indicates whether to display pagination controls in the footer.
pageSizeChanger	number[]	Numeric array for displaying a menu in the pagination for configuring the number of items on each page.
showPreviousNext	boolean	Whether to display previous and next buttons in the pagination.
showInfoSummary	boolean	Whether to show a summary in the pagination.

Prop	Type	Description
paginationClassName	string	Optional class name for the pagination container.
paginationSize	string	Size of the pagination: Available values are: • small • medium (default) • large • null
pagerButtonCount	number	The number of page buttons to display in the pagination controls.
onClick	((event: any, result: string) => void)	Callback function triggered when a result item is clicked.
onMouseEnter	((event: React.SyntheticEvent , result: any) => void)	Callback function triggered when the mouse pointer enters a result item. Receives the event and result objects as arguments.
onMouseLeave	((event: React.SyntheticEvent , result: any) => void)	Callback function triggered when the mouse pointer leaves a result item. Receives the event and result objects as arguments.

ResultsCustom

The ResultsCustom widget displays a list of search results from a `POST /v1/search` response from the MarkLogic REST API. The widget constructs content for each search result using a callback function that is passed a result as a callback argument.

The ResultsCustom widget is similar to the ResultsConfig widget, which maps search results content to the UI based on a configuration object.

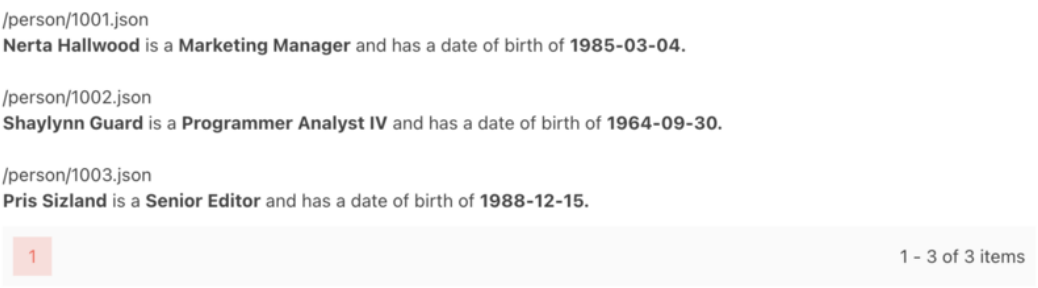
ResultsCustom MarkLogic configuration

To display content from search results documents in the ResultsCustom widget, that content must be extracted and included in the search results. Add an `extract-document-data` property in the query options of the application to do this. See Include document extracts in search results for details.

The ResultsCustom widget expects to receive search results in the `results` prop. The expected format is the format returned from POST /v1/search in the MarkLogic REST API.

ResultsCustom example rendering

In this example, the ResultsCustom widget displays a custom rendering for each search result. The rendering is constructed by the `renderItem` callback.



ResultsCustom example configuration

In this example, the ResultsCustom widget is passed a `renderItem` prop that constructs a custom rendering for each search result. The `myResultRender` function builds the custom rendering using the URI, first name, last name, job title, and date of birth from each search result:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsList, ResultsSnippet } from "ml-
fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const myResultRender = (result, index, handleClick) => {
    const extracted = result?.dataItem?.extracted.content[0].envelope;
    let fullName = extracted.firstName + ' ' + extracted.lastName;
```

```
    return (
      <div
        key={index}
        onClick={handleClick}
        style={{padding: '10px 0', cursor: 'pointer'}}
      >
          <div>{result?.dataItem?.uri}</div>
          <div><strong>{fullName}</strong> is a
<strong>{extracted.title}</strong> and has a date of birth of
<strong>{extracted.dob}.</strong></div>
      </div>
    )
  }

  return (
    <div className="App">
      <div className="SearchBox">
        <SearchBox onSearch={handleSearch}/>
      </div>
      <ResultsList
        results={context.searchResponse.results}
        paginationFooter={true}
        renderItem={(result, index) => myResultRender(result, index, () =>
console.log(result))}
      />
    </div>
  );
}

export default App;
```

ResultsCustom API

Prop	Type	Description
results	object[]	Search results data to display.
renderItem	((result: any, index: number) => ReactElement<any, string | JSXElementConstructor<any>>)	Callback function for rendering each result in the list. Receives a result object and the current index.
title	string	Optional title for the results list.

Prop	Type	Description
titleStyle	CSSProperties	CSS styles applied to the results list title.
className	string	Class name applied to the widget.
headerClassName	string	Class name applied to the widget header.
headerValue	any	Content displayed in the widget header.
footerClassName	number	Class name applied to the widget footer.
footerValue	any	Content to display in the widget footer.
paginationHeader	boolean	Indicates whether to display pagination controls in the header.
paginationFooter	boolean	Indicates whether to display pagination controls in the footer.
pageSizeChanger	number[]	Numeric array for displaying a menu in the pagination for configuring the number of items on each page.
showPreviousNext	boolean	Whether to display previous and next buttons in the pagination.
showInfoSummary	boolean	Whether to show a summary in the pagination.

Prop	Type	Description
paginationClassName	string	Optional class name for the pagination container.
paginationSize	string	Size of the pagination: Available values are: • small • medium (default) • large • null
pagerButtonCount	number	The number of page buttons to display in the pagination controls.

SearchBox

The SearchBox widget displays a text box for entering a MarkLogic string query and a button for submitting the query. It also includes an optional dropdown menu for selecting and submitting a collection constraint.

Example configuration

This example shows a configured SearchBox widget in a React search application:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsSnippet } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
    context.setCollections(params?.collections);
  }

  return (
```

```
    <div className="App">
      <div>
        <SearchBox
          onSearch={handleSearch}
          placeholder="Enter search text"
          menuThemeColor="dark"
          buttonThemeColor="light"
          menuItems={[
            {
              value: ['person', 'organization'],
              label: 'All Entities'
            },
            {
              value: ['person'],
              label: 'Person'
            },
            {
              value: ['organization'],
              label: 'Organization'
            }
          ]}
        />
      </div>
      <div>
        <ResultsSnippet results={context.searchResponse.results} />
      </div>
    </div>
  );
}

export default App;
```

Code explanation

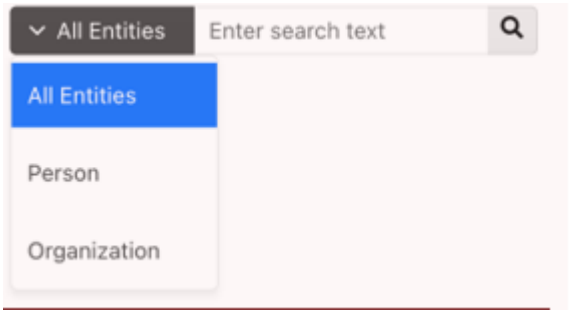
The configured SearchBox widget includes:

- A text input field for typing a search string that is prefilled with a placeholder string.
- A dropdown menu with selections for `All Entities`, `Person`, and `Organization`. To exclude the dropdown menu, do not set a `menuItems` prop.
- A submit button for executing a search.
- A callback function that receives an object with the selected menu item, and the search string when a search is submitted. The search information is set in the application context by the callback function.
- Style settings for the dropdown menu and submit button.

For more information, see API.

Example rendering

The Code explanation displays this:



API

Prop	Type	Description
menuItems	{ label: string; value: string[]; }[]	Array of configuration objects for menu items. Value properties correspond to array of one or more document collections in MarkLogic.
value	string	Default search string.
menuThemeColor	"base" | "primary" | "secondary" | "tertiary" | "info" | "success" | "warning" | "dark" | "light" | "inverse"	The theme color for the menu. See the KendoReact DropDownButtonProps.
menuFillMode	"link" | "solid" | "outline" | "flat"	Kendo fill mode for the menu. See the KendoReact DropDownButtonProps.
buttonThemeColor	"base" | "primary" | "secondary" | "tertiary" | "info" | "success" | "warning" | "dark" | "light" | "inverse"	The theme color variant for the search button. See the KendoReact DropDownButtonProps.
placeholder	string	Placeholder value for the text field.

Prop	Type	Description
className	string	Class name applied to the widget.
ariaLabel	function	Aria label attribute applied to the text field.
containerStyle	CSSProperties	CSS styles applied to the widget.
dropdownButtonStyle	CSSProperties	CSS styles applied to the menu button.
dropdownItemStyle	CSSProperties	CSS styles applied to each menu item.
rightButtonStyle	CSSProperties	CSS styles applied to the search button.
boxStyle	CSSProperties	CSS styles applied to the text field.
selected	number	Index of the element in the menu items array to set by default.
disabled	boolean	Whether the text field is disabled. The default is false.
searchSuggest	boolean	Turn on typeahead search suggestions.
searchSuggestMin	number	Minimum number of characters required before suggestions are shown when `searchSuggest` is turned on. If unspecified, the default number of characters needed before suggestions appear is 1.

Prop	Type	Description
searchSuggestSubmit	boolean	Indicates whether selecting a search suggestion from the menu will automatically submit a query.
searchSuggestLimit	number	Optional numeric setting for limiting the number of suggestions being returned. Default is 10.
showLoading	boolean	Display a loading indicator while search suggestions are requested.
selectedLabel	string	String label to set in the menu by default.
onChange	((event: AutoCompleteChangeEvent) => void)	Callback function triggered by a change in the text field.
onChangeMenu	((menuIndex: number) => void)	Callback function triggered by a change in the menu.
onClick	((event: MouseEvent<HTMLButtonElement, MouseEvent>) => void)	Callback function triggered by a search button click. Called before `onSearch`.
onEnter	((event: KeyboardEvent<HTMLInputElement>) => void)	Callback function triggered by a keyboard enter event on text field. Called before `onSearch`.
onSearch	((params: { q: string; collections: string[]; }) => void)	Callback function for a search-button click or text-field enter event. Called after `onClick` and `onEnter`.

SelectedFacets

The SelectedFacets widget displays colored, labeled tags representing the facet values selected in a search application. The widget can display selections from these FastTrack widgets:

- StringFacet
- DateRangeFacet
- BucketRangeFacet
- NumberRangeFacet

Example configuration

This React application code includes the SelectedFacets widget. The code configures the widget to display current selections from the four types of facet selection widgets.

```
import { SelectedFacets } from "ml-fasttrack"

const App = () => {

  const [valueDateFacet, setValueDateFacet] = useState({ start: new
Date(1980, 1, 1), end: new Date(2020, 12, 31) });
  const [resetNumberFacet, setResetNumberFacet] = useState(false);
  const [resetBucketFacet, setResetBucketFacet] = useState('');
  const [resetMultistringFacet, setResetMultistringFacet] = useState('');
  const [resetStringFacet, setResetStringFacet] = useState('');

  const removeFacets = (facet, type, value) => {
    if (type === 'rf') {
      // Number range facet
      if (facet[0]?.type === 'number') {
        setResetNumberFacet(true)
      } else {
        // Date range facet
        setValueDateFacet({ start: null, end: null })
      }
      context.removeRangeFacetConstraint(facet)
    }
    else if (type === 'sf') {
      // Bucketed facet
      if (facet?.name === 'bucketedString') {
        setResetBucketFacet(facet?.value[0])
        context.removeStringFacetConstraint(facet)
      // Multi-select string
      } else if (facet?.name === 'multiString') {
        setResetMultistringFacet(value)
        context.removeStringFacetConstraint(facet, value)
      } else {
        // String facet
        setResetStringFacet(facet?.value[0])
        context.removeStringFacetConstraint(facet)
```

```
      }
    }
  }

  return (
    <SelectedFacets
      selectedFacetConfig={{
        'string': {
          color: 'red',
          closable: true
        },
        'date': {
          color: 'blue',
          closable: true,
          iconLabel: <i className='fas fa-calendar' style={{ marginRight: 3
}}></i>
        },
        'number': {
          color: 'green',
          dashed: false,
          closable: true,
          iconLabel: <i className='fas fa-dollar-sign' style={{ marginRight:
3 }}></i>
        },
      }}
      stringFacets={context.stringFacetConstraints}
      rangeFacets={context.rangeFacetConstraints}
      removeStringFacet={(f, v) => removeFacets(f, 'sf', v)}
      removeRangeFacet={(f) => removeFacets(f, 'rf')}
      separator="to"
    ></SelectedFacets>
  )

}
```

Code explanation

The SelectedFacets widget displays tags based on the selected facet values set in the application context.

- String facet selections and range facet selections are stored separately in the `stringFacetConstraints` and `rangeFacetConstraints` context variables. As those values change, the set of displayed tags change.
- The `removeStringFacet` and `removeRangeFacet` callback functions handle close events.
- The `removeFacets` function determines the type of facet being closed.
- The `removeStringFacetConstraint` or `removeRangeFacetConstraint` methods handle the closure in the application context.
- The example code also removes the facet selections from the facet selections widgets (StringFacet,

DateRangeFacet, BucketRangeFacet, or NumberRangeFacet).

> **Note:**
>
> The StringFacet, DateRangeFacet, BucketRangeFacet, and NumberRangeFacet widgets are not shown in the Example Configuration.

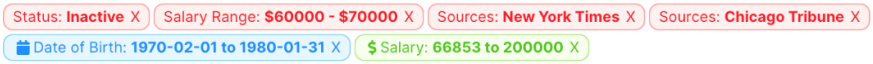
- The local state variables (`valueDateFacet`, `resetNumberFacet`, `resetBucketFacet`, `resetMultistringFacet`, `resetStringFacet`) handle the reset state for the facet widgets. See StringFacet, DateRangeFacet, BucketRangeFacet, or NumberRangeFacet for details.

> **Note:**
>
> In the Example Configuration, the multi-string example represents a string facet that can have multiple value selections at once (hence the value argument is required when calling `removeStringFacetConstraint`). The string example represents a string facet that can only have a single selection at a time (hence the value is not required when calling `removeStringFacetConstraint`).

- The `selectedFacetConfig` object handles styling and other features. See API for more information.

Example rendering

This example shows the tags rendered by the SelectedFacets widget:



The Status facet is an example of a string facet that can only have a single selection. The Sources facet is an example of a string facet that can have multiple selections.

API

Prop	Type	Description
stringFacets	object[]	Array of selected string facets.

Prop	Type	Description
removeStringFacet	((facet?: any, value?: string) => void)	Callback function triggered when the close icon is clicked on string facet tags.
rangeFacets	object[]	Array of selected range facets.
removeRangeFacet	((facet?: any) => void)	Callback function triggered when the close icon is clicked on range facet tags. The function receives a range facet object.
iconLabel	ReactNode	Element displayed to the left of the tag label.
ariaLabel	string	Component "aria-label" value.
color	"green" | "black" | "blue" | "grey" | "magenta" | "red" | "yellow"	Color of the tag and label.
closable	boolean	Indicates whether to display the close icon and handle the close events with the `onClose` callback.
dashed	boolean	Indicates whether to add a dashed style to the tag.
id	string	Widget `id` value.
label	ReactNode	Label of the element to show. Overrides the facet of the tag.
style	CSSProperties	CSS styles applied to the tag.

Prop	Type	Description
className	string	Class name applied to the tag.
visible	boolean	Indicates whether to show the widget.
separator	string	String separator for range values.
closeIcon	ReactNode	Element to use as the close icon.
classNameCloseIcon	string	Class name applied to the close icon.
selectedFacetConfig	Record<string, { color?: string; dashed?: boolean; closable?: boolean; visible?: boolean; closeIcon?: ReactNode; iconLabel?: ReactNode; }>	Optional object to handle the facet properties (color, dashed, closable, visible, close icon, iconLabel) by type of facet, e.g.: string, number, date.

Slider

The Slider enables users to select a numeric value within a range by clicking and dragging a handle along a horizontal line. A callback function can use that value to define a constraint for a MarkLogic search.

Slider example rendering

In this example, the Slider displays a draggable handle with a prefix (**0**), an input box, and a suffix (**miles**).

Slider example configuration

In this example, the Slider is configured to support a range from 0 to 100. An `onChange` event handler sets a

state variable to the Slider value when the Slider changes:

```
function App() {

    const [sliderValue, setSliderValue] = useState(0);

  const handleChange = async (val) {
    setSliderValue(val)
    console.log('Slider changed to', val)
  }

  return (
    <div className="App">
      <Slider
        min={0}
        max={100}
  prefix={'0'}
    suffix={'miles'}
  defaultValue={0}
        onChange={handleChange}
      />
    </div>
  );
}

export default App;
```

Slider API

Prop	Type	Description
prefix	string	Optional string added to the left of the widget.
suffix	string	Optional string added to the right of the widget.
min	number	Minimum value of the Slider.
max	number	Maximum value of the Slider.
defaultValue	number	Default value of the Slider.

Prop	Type	Description
showInput	boolean	Indicates whether to show a numeric input box.
sliderSettings	Record<string, any>	Object of prop values passed to the KendoReact Slider.
inputSettings	Record<string, any>	Object of prop values passed to the KendoReact Numeric Text Box.
onChange	(sliderVal: number) => void	Callback function triggered when the Slider changes. Receives the numeric Slider value.

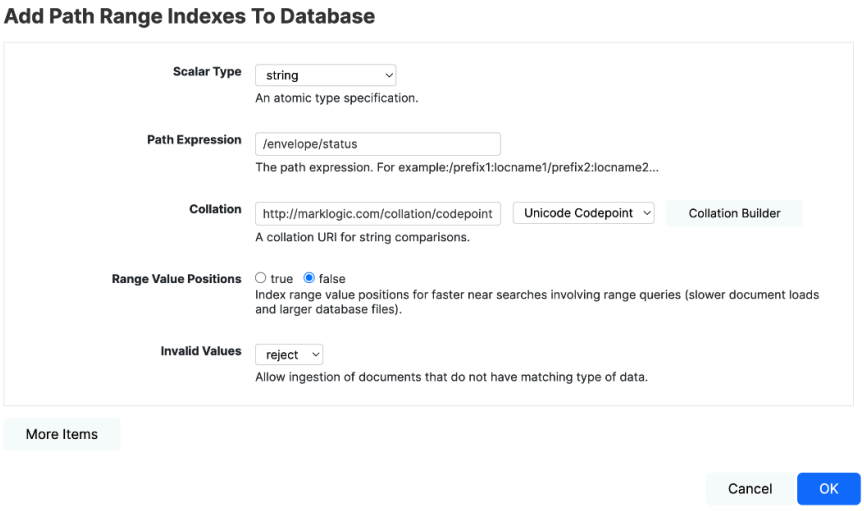
StringFacet

The StringFacet widget displays a summary of values and counts for a faceted property in search results. Users can constrain a search by a facet value by clicking a check box.

StringFacet MarkLogic setup

Faceted search in MarkLogic requires a range index on the faceted property. Add a range index for a property to the database configuration using the Admin Interface or one of MarkLogic's programmatic APIs.

This example shows a path range index added to the `status` property in the Admin Interface:



After the index is setup, a constraint can be configured in the query options for the search application. The constraint returns facets for the property in the search results. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="status">
    <range type="xs:string" facet="true" collation="http://marklogic.com/
collation/codepoint">
      <path-index>/envelope/status</path-index>
      <facet-option>limit=25</facet-option>
      <facet-option>frequency-order</facet-option>
      <facet-option>descending</facet-option>
    </range>
  </constraint>
  <return-facets>true</return-facets>
</options>
```

The constraint settings correspond to the settings for the index. `return-facets` is set to `true` so that facet results are returned with search results. This example shows the `status` facet information returned in the search response:

```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "score": 0,
            "confidence": 0,
            "fitness": 0,
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "mimetype": "application/json",
            "format": "json",
            "matches": [
                {
                    "path": "fn:doc(\"/person/1001.json\")/object-node()",
                    "match-text": [
                        "person Nerta Hallwood Marketing Manager Active
1985-03-04 person-1001.jpg"
                    ]
                }
            ],
            "extracted": {
                "kind": "array",
                "content": [
                    {
```

```
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
                            "status": "Active",
                            "dob": "1985-03-04",
                            "image": "person-1001.jpg",
                        }
                    }
                ]
            }
        },
        // ...
    ],
    "facets": {
        "status": {
            "type": "xs:string",
            "facetValues": [
                {
                    "name": "Inactive",
                    "count": 52,
                    "value": "Inactive"
                },
                {
                    "name": "Active",
                    "count": 48,
                    "value": "Active"
                }
            ]
        }
    },
    // ...
}
```

StringFacet example rendering

This example shows the StringFacet widget rendering values for a `status` property from a set of documents. Documents in the set have status values of either **Active** or **Inactive**. Clicking a value applies the constraint for the facet. After a value is clicked, the UI is updated to display results with the constraint applied. In this example, only documents with the `status` property set to **Active** will be returned and displayed.

StringFacet example configuration

This example shows how to import and configure the StringFacet widget in a React application:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, ResultsSnippet, StringFacet } from "ml-
fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  const handleFacetClick = (selection) => {
    context.addStringFacetConstraint(selection)
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div style={{display: 'flex', flexDirection: 'row'}}>
          <div style={{width: '640px'}}>
            <ResultsSnippet
              results={context.searchResponse.results}
              paginationFooter={true}
            />
          </div>
          <div>
            {context.searchResponse?.facets?.status &&
              <StringFacet
                title="Status"
                name="status"
                data={context.searchResponse?.facets?.status}
```

```
                onSelect={handleFacetClick}
              />
            }
          </div>
        </div>
      </div>
    );
}

export default App;
```

Code explanation

In the StringFacet example configuration:

- The facet in the `data` prop is the facet displayed from the search response object.
- The `onSelect` callback handles check box clicks and receives a selection object from the widget. It can then set the facet constraint in the application context using the `addStringFacetConstraint` method.

StringFacet API

Prop	Type	Description
data	{ type: string; facetValues: FacetValue[]; }	Facet data to display from search results.
title	string	Facet title for the collapsible header.
subTitle	String	Facet subtitle for the collapsible header.
name	string	String identifying the facet. Passed as the name value in the `onSelect` event.
threshold	number	The number of items to display before showing the show more option.

Prop	Type	Description
containerStyle	CSSProperties	CSS styles applied to the widget.
reset	string	String value identifying the item to uncheck. For integration with the SelectedFacets widget.
onSelect	(value: { type: string; name: string; value: string[]; title?: string | undefined; }) => void	Callback function triggered by a select event. Receives a selection object.

StringFacet callback argument

This example shows a selection object passed to the `onSelect` callback:

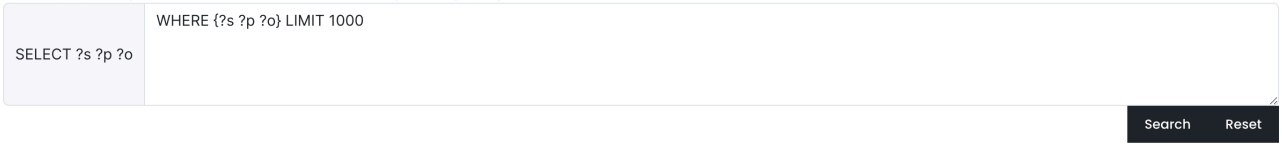
```
{
    "type": "string",
    "name": "status",
    "value": [
        "Active"
    ],
    "title": "Status"
}
```

SemanticQuery

The SemanticQuery widget queries semantic data from a database. An input field accepts native SPARQL queries. When a query is executed, it retrieves information from RDF triples in the data. The search results are returned with an array of data objects that contain information on the corresponding triple. The data is organized into subject, predicate, and object form. The SemanticQuery widget typically works alongside the NetworkGraph widget. The NetworkGraph widget displays connected nodes based on search results with triples.

Example rendering

This is an example of the `SemanticQuery` widget.



Explanation

In the Example rendering:

- The SELECT clause identifies the three variables (s, p, o) that appear in the query results. The appearance of individual variables can be turned on or off using the widget props. See API for additional information. The remainder of the query can be modified freely through the input field.
- The WHERE clause provides a basic pattern to match against the data.
- The LIMIT sets the maximum number of results to 1000.
- Clicking the **Search** button submits the query with a callback specified through the widget props,
- The **Reset** button resets the query back to its default state.

Example search response

This response shows triples data in subject-predicate-object form. In the example, `s` refers to the subject, `p` refers to the predicate, and `o` refers to the object.


```
{
    "head": {
        "vars": [
            "s",
            "p",
            "o"
        ]
    },
    "results": {
        "bindings": [
            {
                "s": {
                    "type": "uri",
                    "value": "http://example.org/organization/10029.xml "
                },
                "p": {
                    "type": "uri",
                    "value": "http://example.org/employs"
                },
                "o": {
                    "type": "uri",
                    "value": "http://example.org/person/10034.xml"
                }
```

```
            }
        ]
    }
}
```

Example configuration

This example React application displays a SemanticQuery widget. The widget submits a semantic query through the `getSparql()a` method in `MarkLogicContext`. The method stores the results in a `sparqlResponse` state object. After the query is submitted, the object is transformed to display a summary of the results in the NetworkGraph widget.

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SemanticQuery } from "ml-fasttrack";
import { transformSparqlResult } from './config/util.config.js';

  function App() {

  const context = useContext(MarkLogicContext);

  const handleSemanticQuery = (query) => {
    context.getSparql(query);
  };
  const transformSparqlResult = (data) => {
    if (!data) return;
    let transformed = {};
    data.results.bindings.forEach(r => {
      const { s, p, o } = r
      if (s && s?.value) {
        if (!transformed[s.value]) {
          transformed[s.value] = {
            label: [{ text: s?.value, position: 's' }],
            uri: s?.value
          }
        }
      }
      if (o && o?.value) {
        if (!transformed[o.value]) {
          transformed[o.value] = {
            label: [{ text: o.value, position: 's' }],
            uri: o.value
          }
        }
      }
      if (s && s?.value && o && o?.value) {
        if (!transformed[s.value + '-' + o.value]) {
```

```
          transformed[s.value + '-' + o.value] = {
            id1: s.value,
            id2: o.value,
            label: {
              text: p?.value
            }
          }
        }
      }
    })
    return transformed;
  }

  const sparqlItems = transformSparqlResult(context.sparqlResponse)

  return (
    <div className="App">
      <div>
        <SemanticQuery buttonVariant="dark" onSearch={handleSemanticQuery}
inputRows={4} />
        <div>
              Found {sparqlItems ? Object.keys(sparqlItems).filter(key =>
!key?.includes('-'))?.length : 0} results
        </div>
        <div>
          <NetworkGraph
            items={ sparqlItems }
            width={ '100%' }
            height={'600px' }
          />

        </div>
      </div>
    </div>
  );
}

export default App;
```

API

Prop	Type	Description
inputRows	number	Number of rows displayed for the input text box.

Prop	Type	Description
disableSource	boolean	Indicates whether `?s` Is disabled in the select query.
disablePredicate	boolean	Indicates whether `?p` is disabled in the select query.
disableObject	boolean	Indicates whether `?o` is disabled in the select query.
buttonVariant	"light" | "dark" | "base" | "primary" | "secondary" | "tertiary" | "info" | "success" | "warning" | "error" | "inverse"	KendoReact theme color for the buttons.
boxVariant	"light" | "dark" | "base" | "primary" | "secondary" | "tertiary" | "info" | "success" | "warning" | "error" | "inverse"	KendoReact theme color for the side box.
containerStyle	CSSProperties	CSS styles applied to the widget.
textAreaStyle	CSSProperties	CSS styles applied to the text area.
inputGroupStyle	CSSProperties	CSS styles applied to the input group container.
actionButtonStyle	CSSProperties	CSS styles applied to the action buttons container.
onSearch	((sparqlQuery: string) => void)	Callback function triggered when the search button is clicked.

Timeline

The Timeline widget displays time-based information (events, activities, etc.) for one or more entity instances along a timeline.

Timeline MarkLogic setup

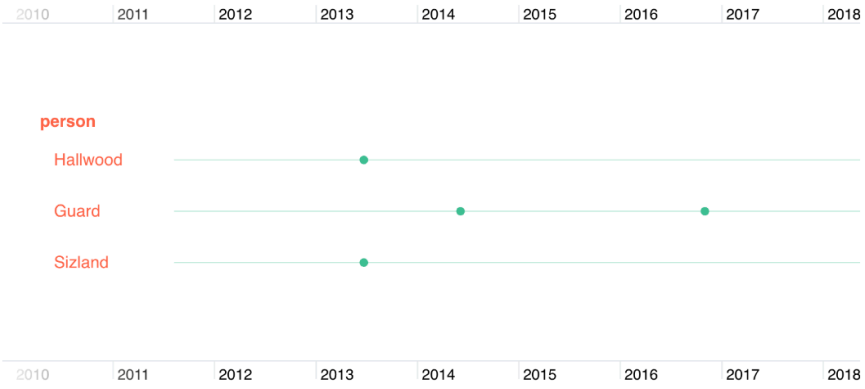
To display time-based information from `/v1/search` results in the Timeline widget, the content must be extracted and included in the search results. To do this, put an `extract-document-data` property in the query options of the application. In the example, the document content is shown under an `extracted` property. See Include document extracts in search results for details.

Timeline concepts

The Timeline widget displays event markers along a timeline based on the marker's datetime values. Each event is associated with an entity instance. Multiple events for an instance are placed along the same row in the timeline. A label for each entity is shown on the left-side of the timeline.

The set of events to include in the timeline is specified from the data source. The Timeline widget automatically sizes itself to display all the events on load. Labels are displayed for the different entity instances that have events currently shown.

Example rendering



To change the timeline's range, click and drag along the timeline. Tablet or trackpad users can also use the spread and pinch gestures or commands.

Show time-based events from search results

The Timeline widget can map event information from a `/v1/search` response onto a timeline. The events for each result in the search response are displayed in a timeline row.

Example response

This example shows a `/v1/search` response payload. Each result in the response has an array of event information in an `activities` property. These events can be displayed on a timeline and associated with their entity instance (which in this case is a person).

```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
                            "status": "active",
                            "activities": [
                                {
                                    "description": "Started at Fadeo",
                                    "ts": "2013-06-22"
                                },
                                {
                                    "description": "Promoted",
                                    "ts": "2019-08-15"
                                }
                            ]
                        }
                    }
                ]
            }
        },
        // more results...
    ],
    // more metadata...
}
```

Example React application

This React application includes the Timeline widget configured to display the example payload:

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, SearchBox, Timeline } from "ml-fasttrack";

function App() {

  const context = useContext(MarkLogicContext);

  const TimelineConfig = {
    entityTypeConfig: {
      path: "extracted.content[0].envelope.entityType"
    },
    entities: [
      {
          entityType: 'person',
          path: 'extracted.content[0].envelope',
          label: 'lastName',
          eventPath: 'activities',
          items: [
              {
                  label: 'description',
                  timePath: 'ts'
              },
          ],
          detailConfig: [
              {
                  label: 'First Name',
                  path: 'firstName'
              },
              {
                label: 'Last Name',
                path: 'lastName'
              },
          ]
      }
    ]
  }

  const TimelineSettings = {
    entityTypes: {
        default: {
            labelColor: 'white'
        },
        person: {
            labelColor: 'tomato'
        },
```

```
        organization: {
            labelColor: '#D4886A'
        }
    },
    eventTypes: {
            default: {
                showArrows: true,
            },
            "2013-06-22": {
                color: 'fuchsia'
            }
        }
    }

  const handleSearch = (params) => {
    context.setQtext(params?.q);
  }

  return (
    <div className="App">
      <div>
        <SearchBox onSearch={handleSearch}/>
      </div>
      <div>
          <div style={{height: '400px'}}>
          <Timeline
            data={context?.searchResponse?.results}
            config={TimelineConfig}
            theme={'light'}
            onTimelineClick={(event) => console.log('Timeline!', event)}
            settings={TimelineSettings}
          />
          </div>
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In the example:

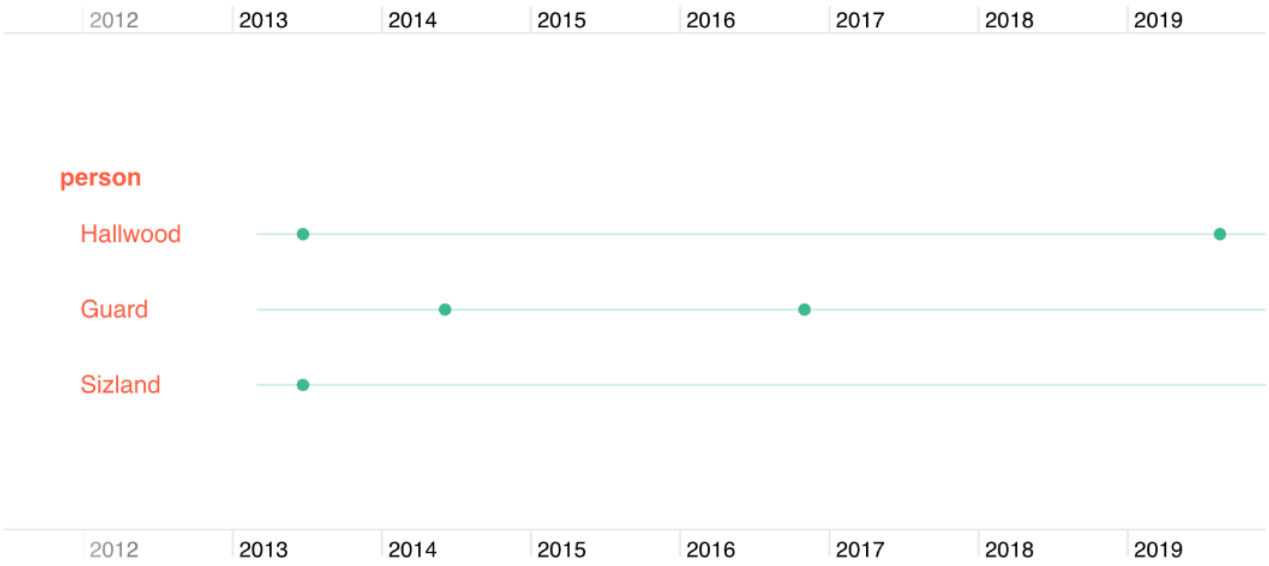
- A user executes a search using a FastTrack SearchBox widget and the search response is stored in the application context. Results from the search response are assigned to the Timeline widget's `data` prop.
- The event data from the `data` prop is mapped using the `config` prop.
- The `entityTypeConfig` property in `config` specifies the `path` to the entity type in each result. In this example, `"person"` is the only entity type in the results.

- The `entities` array in `config` maps the timeline information from the `data` prop using one or more configuration objects. One object is used for each entity type. For details about the configuration objects, see the API.
- The `settings` prop can be used to specify styles for the entity and event information in the timeline. For details, see Styling Entities and Events, API, and the third-party documentation included with FastTrack.

The Timeline widget supports event handlers corresponding to user interactions on the timeline. In the example, the `onTimelineClick` event handler responds to clicks on timeline events. For details about supported event handlers, see API.

Timeline example rendering

The Example React application renders this:



Showing time-based events from a document

Time-based information can be mapped from a `/v1/document` result and displayed by the Timeline widget. The result shows events stored as an array of objects in an `activities` property. These events can be displayed on a timeline for that person.

Example response

This is an example response for a person document:

```
{
    "snippet-format": "snippet",
    "total": 3,
    "start": 1,
    "page-length": 10,
```

```
    "selected": "include",
    "results": [
        {
            "index": 1,
            "uri": "/person/1001.json",
            "path": "fn:doc(\"/person/1001.json\")",
            "href": "/v1/documents?uri=%2Fperson%2F1001.json",
            "extracted": {
                "kind": "array",
                "content": [
                    {
                        "envelope": {
                            "entityType": "person",
                            "id": 1001,
                            "firstName": "Nerta",
                            "lastName": "Hallwood",
                            "title": "Marketing Manager",
                            "status": "active",
                            "activities": [
                                {
                                    "description": "Started at Fadeo",
                                    "ts": "2013-06-22"
                                },
                                {
                                    "description": "Promoted",
                                    "ts": "2019-08-15"
                                }
                            ]
                        }
                    }
                ]
            }
        },
        // more results...
    ],
    // more metadata...
}
```

React application code

This is a React application with the Timeline widget configured to display the data from the response:

```
import { useContext, useEffect } from "react";
import './App.css';
import { MarkLogicContext, Timeline } from "ml-fasttrack";

function App() {
```

```
  const context = useContext(MarkLogicContext);

  useEffect(() => {
    context.getDocument('/person/1001.json');
  }, []);

  const TimelineConfig = {
      entityTypeConfig: {
        path: "data.envelope.entityType"
      },
      entities: [
        {
            entityType: 'person',
            path: 'data.envelope',
            label: 'firstName',
            eventPath: 'activities',
            items: [
                {
                    label: 'description',
                    timePath: 'ts'
                },
            ]
        }
      ]
  }

  return (
    <div className="App">
      <div>
          <div style={{height: '240px'}}>
          <Timeline
            data={context?.documentResponse}
            config={TimelineConfig}
          />
          </div>
      </div>
    </div>
  );
}

export default App;
```

Code explanation

In the example application:

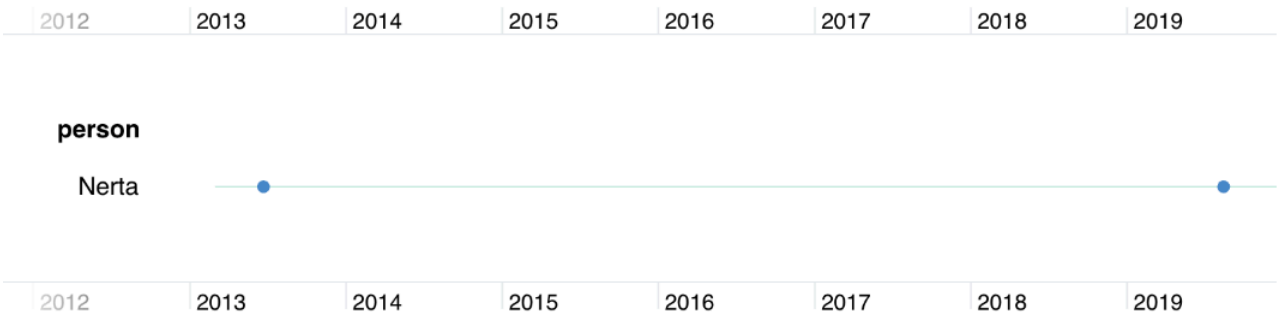
- A `useEffect` React hook loads a document using the `getDocument` method in the application content (which makes a call to the `/v1/documents` endpoint). The document response is stored in the application context.
- The document response is assigned to the Timeline widget's `data` prop. The event data from the `data`

prop is mapped using the `config` prop.

- The `entityTypeConfig` property in `config` specifies the `path` to the entity type in the document response.
- The `entities` array in `config` allows the timeline information from the `data` prop to be mapped using a configuration object for the document. For details about the configuration object, see API.
- The `settings` prop specifies styles for the entity and event information in the timeline. For details, see Styling Entities and Events, Timeline API, and the third-party documentation included with FastTrack.

Timeline example rendering

The Timeline widget configured with the example code renders this:



A row represents the events in the document.

Styling entities and events

Style entities and events in the timeline by adding a Timeline `settings` prop. The `settings` prop can be used to:

- change the color of the entity labels.
- change the color and width of the entity line.
- change the colors of the event markers.
- use icons with events.

Styling entities and events example code

The `settings` prop accepts an `entityTypes` property for specifying entity styles and an `eventTypes` property for specifying event styles. This code defines a `TimelineSettings` variable to use for the `settings` prop:

```
const TimelineSettings = {
    entityTypes: {
```

```
        Persons: {
          labelColor: "darkslategray",
          color: "purple",
          lineWidth: 6
        }
    },
    eventTypes: {
        default: {
            fontIcon: {
              fontFamily: "Font Awesome 5 Free",
              fontWeight: 900,
              text: "\uf101"
            }
        },
        "Started at Fadeo": {
            color: "orange"
        },
        Promoted: {
            color: "fuchsia"

        }
    }
}
```

> **Note:**
>
> A `default` property key can be used in the `entityTypes` and `eventTypes` objects to set default styles across all the entities and events in the timeline. The example includes a `default` key in the `eventTypes` object to set an icon to use for all the events in the timeline.

React application code

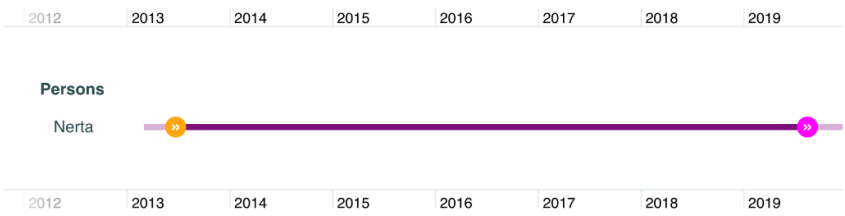
This is sample React application code when the `settings` prop is added to the Timeline widget:

```
import '@fortawesome/fontawesome-free/css/fontawesome.css';
import '@fortawesome/fontawesome-free/css/all.css';

<Timeline
  data={context?.documentResponse}
  config={TimelineConfig}
  settings={TimelineSettings}
/>
```

> **Note:**
>
> The import statements are required to load the Font Awesome library referenced in the `settings` prop. The Font Awesome library must be installed as a dependency in the `package.json` file of the React application.

Styling entities and events example rendering

Styling the entities and events using the `settings` prop described in Styling Entities and Events renders this:



For more information, see the third-party documentation included with FastTrack.

Configuring event popups

A popup window can be configured to open when an event in the timeline is clicked. The content that appears in the window is defined by adding a `detailConfig` array to the entity configuration object.

Example popup code

This example shows a `detailConfig` array. The array defines a popup window that displays the first and last name of the person entity instance associated with the event:

```
const TimelineConfig = {
    entityTypeConfig: {
      path: "data.envelope.entityType"
    },
    entities: [
      {
          entityType: 'person',
          path: 'data.envelope',
          label: 'firstName',
          eventPath: 'activities',
          items: [
              {
                  label: 'description',
                  timePath: 'ts'
```

```
            },
        ],
        detailConfig: [
            {
                label: "First Name",
                path: "firstName"
            },
            {
                label: "Last Name",
                path: "lastName"
            }
        ]
    }
  ]
}
```

Code explanation

For each `detailConfig` array element, the window displays a label followed by the value determined by a path configuration (which is relative to the parent path value).

The popup window is based on the Kendo Window component.

Style the window

The window can be styled by passing a `windowSettings` prop into the Timeline widget. This configuration object sets the window title, window height and width, and also makes the window draggable:

```
onst WindowConfig = {
  title: "Event Info",
  initialHeight: 150,
  initialWidth: 240,
  draggable: true
}
```

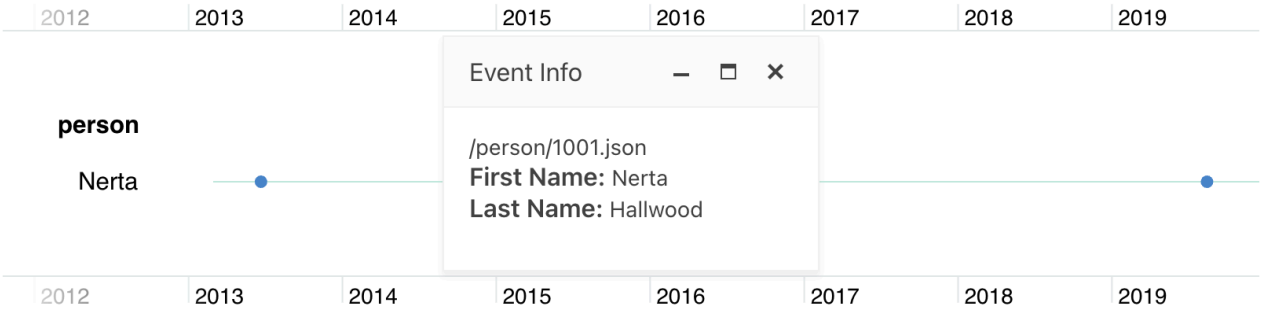
React application code

The React application code looks like this with the popup window configuration added to the `windowSettings` prop for the Timeline widget:

```
<Timeline
  data={context?.documentResponse}
  config={TimelineConfig}
  windowSettings={WindowConfig}
/>
```

Example rendering

Adding the popup window configuration information results in this window appearing when an event is clicked:



Timeline API

Prop	Type	Description
data	object	Data to display in the widget.
config	object	Timeline configuration object. The `entityTypeConfig` object specifies the path to the entity type in the data. The array of objects in the entity array determines how Timeline data is mapped for each entity type.
theme	string	Timeline theme (“`light`” or “`dark`”).
containerStyle	CSSProperties	CSS styles applied to the widget.
settings	Record<string, any>	Props passed in for configuring the timeline. For more information, see the third-party documentation included with FastTrack.

Prop	Type	Description
onTimelineHover	((event: Record<string, any>) => void)	Callback function triggered when the user hovers over an item. Receives an event object. For more information, see the third-party documentation included with FastTrack.
onTimelineDragStartHandler	((event: Record<string, any>) => void)	Callback function triggered when a drag is started. Receives an event object. For more information, see the third-party documentation included with FastTrack.
onTimelineClick	((event: Record<string, any>) => void)	Callback function invoked when the user clicks the timeline. Receives an event object.
windowSettings	Record<string, any>	Callback function triggered when the user hovers over an item. Receives an event object. For more information, see the third-party documentation included with FastTrack.
windowDetailOff	boolean	Indicates whether to show the detail window on event click.
showTooltip	boolean	Indicates whether to show a tooltip on event hover. The `detailConfig` setting determines the tooltip content.

config API

Property	Type	Description
entityTypeConfig	PathConfig	An entity type configuration object.
entityTypeConfig.path	string	The path to the entity type in the search result. The path is specified using JSONPath.
entities[]	object[]	An array of graph configuration objects for each entity.
entities[].entityType	string	The entity type of the configuration object.
entities[].path	string	Path to the timeline data in the payload. The path is specified using JSONPath.
entities[].label	string	The path to the entity label for each timeline item relative to the path to the timeline data. The path is specified using JSONPath.
entities[].eventPath	string	The path to the event data relative to the path to the timeline data. The path is specified using JSONPath.
entities[].items	object[]	One or more configuration objects for the events to display in the timeline.
entities[].items.label	string	The path to the event label relative to the path to the event data. The path is specified using JSONPath.
entities[].items.timePath	string	The path to the event timestamp relative to the path to the event data. The path is specified using JSONPath.
entities[].detailConfig	object[]	One or more configuration objects for the content to display in the event popup.

Property	Type	Description
entities[].detailConfig.label	object	The content label for the value.
entities[].detailConfig.path	string	The path to the value relative to the path to the timeline data. The path is specified using JSONPath.

settings API

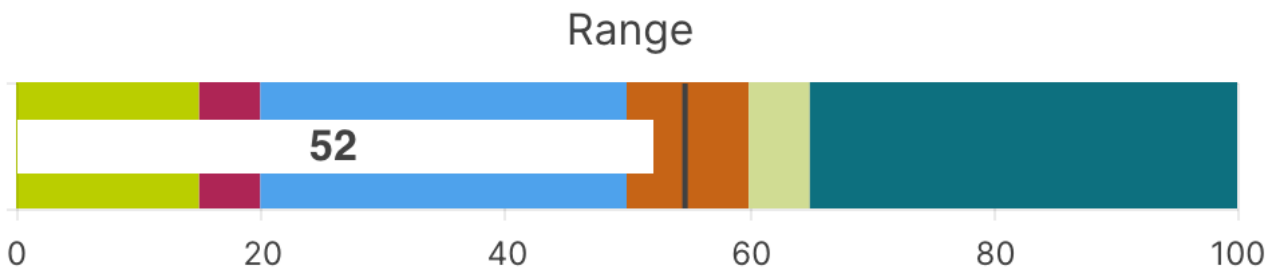
Property	Type	Description
entityTypes	object	Configuration settings for different entity types. Use default for default entity settings. For more information, see the third-party documentation included with FastTrack.
eventTypes	object	Configuration settings for different events in the timeline. Use default for default event settings. For more information, see the third-party documentation included with FastTrack.

TwoLayersChart

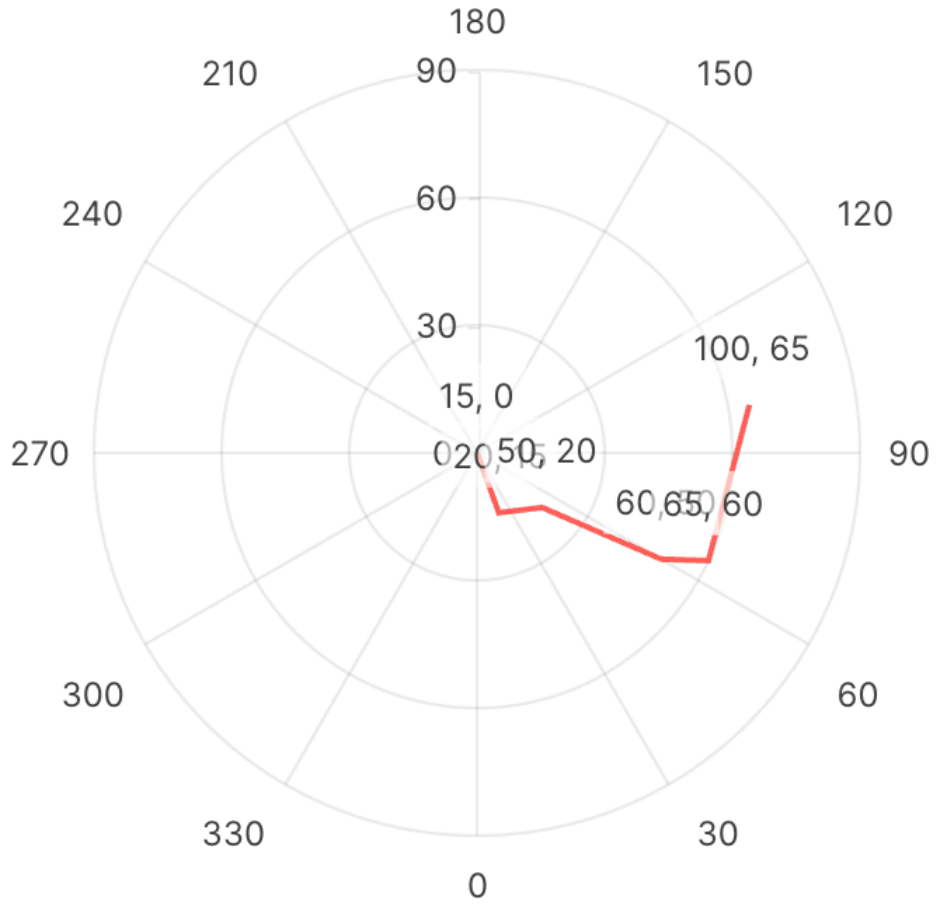
The TwoLayersChart widget displays multivariate document information in a variety of charts. The TwoLayersChart widget is based on the KendoReact Chart component. Many of KendoReact Chart settings can be passed to configure the TwoLayersChart. See API, config API, and *ChartProps API.

Supported charts

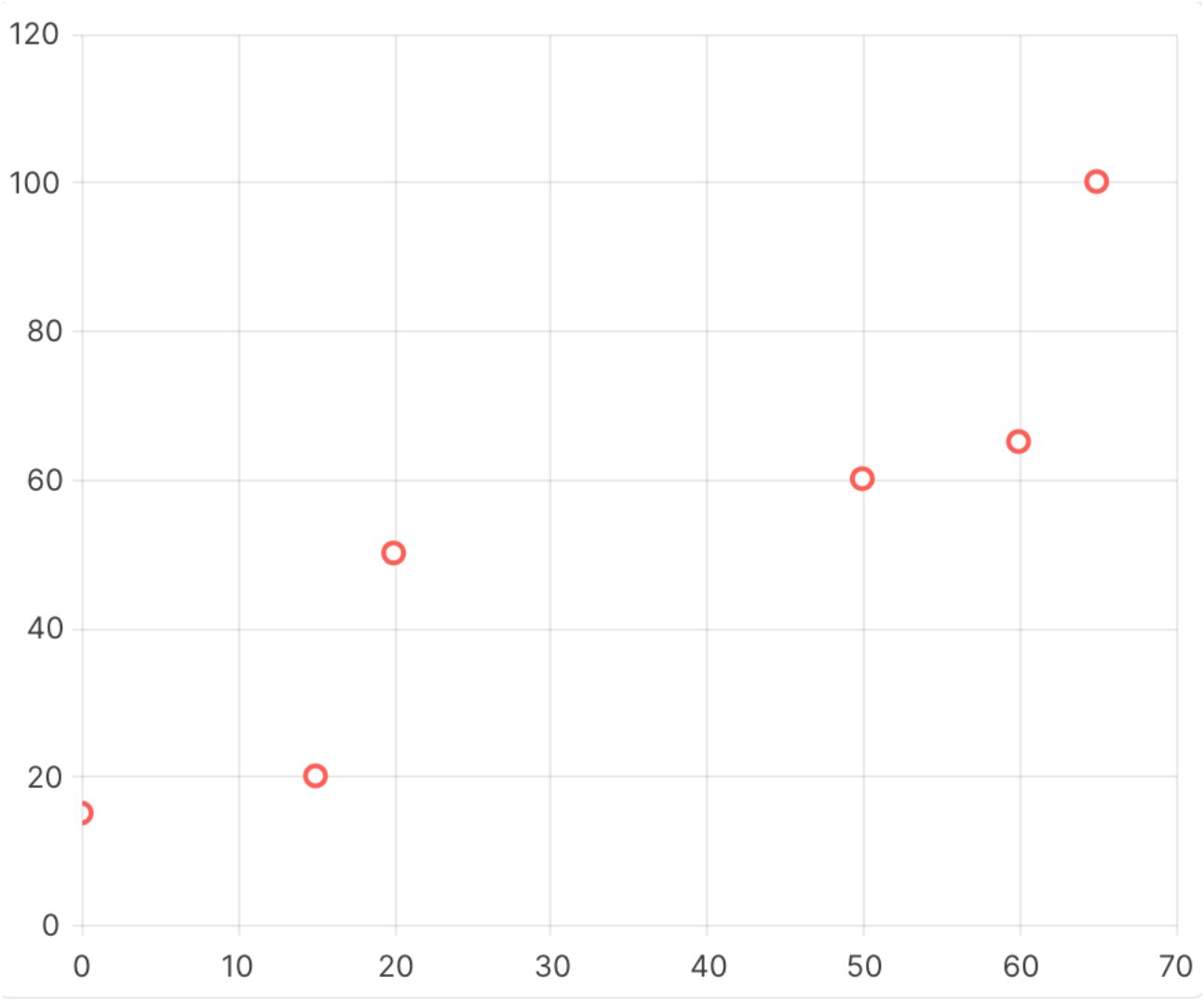
Bullet chart



Polar chart



Scatterplot chart



Example data

The Bullet chart and Polar chart examples use this search result data:

```
const exampleData = [
    {
        "index": 1,
        "uri": "/person/10054.xml",
        "path": "fn:doc(\"/person/10054.xml\")",
        "extracted": {
            "person": {
                "personId": 10054,
                "range": { "from": 0, "to": 15 },
                "polar": { "time": "08:00", "altitude": 4.9, "azimuth": 92.7
}
```

```
            }
        }
    },
    {
        "index": 2,
        "uri": "/person/10089.xml",
        "path": "fn:doc(\"/person/10089.xml\")",
        "extracted": {
            "person": {
                "personId": 10089,
                "range": { "from": 15, "to": 20 },
                "polar": { "time": "09:00", "altitude": 6.5, "azimuth": 95.7
}
            }
        }
    },
    {
        "index": 3,
        "uri": "/person/10093.xml",
        "path": "fn:doc(\"/person/10093.xml\")",
        "extracted": {
            "person": {
                "personId": 10093,
                "range": { "from": 20, "to": 50 },
                "polar": { "time": "10:00", "altitude": 7.2, "azimuth": 96.4
}
            }
        }
    },
    {
        "index": 4,
        "uri": "/organization/20001.xml",
        "path": "fn:doc(\"/organization/20001.xml\")",
        "extracted": {
            "organization": {
                "organizationId": 20001,
                "range": { "from": 50, "to": 60 },
                "polar": { "time": "12:00", "altitude": 8.0, "azimuth": 96.7
}
            }
        }
    },
    {
        "index": 5,
        "uri": "/organization/20002.xml",
        "path": "fn:doc(\"/organization/20002.xml\")",
        "extracted": {
            "organization": {
                "organizationId": 20002,
                "range": { "from": 60, "to": 65 },
                "polar": { "time": "13:00", "altitude": 8.3, "azimuth": 97.0
}
```

```
            }
        }
    },
    {
        "index": 6,
        "uri": "/organization/20003.xml",
        "path": "fn:doc(\"/organization/20003.xml\")",
        "extracted": {
            "organization": {
                "organizationId": 20003,
                "range": { "from": 65, "to": 100 },
                "polar": { "time": "13:30", "altitude": 8.5, "azimuth": 97.7
}
            }
        }
    }
]
```

Display a bullet chart from search results

The bullet chart is a variation of a bar chart. A bullet chart compares a quantitative measure (such as temperature) against a qualitative range (such as warm, hot, and cold). A symbol marker is used to encode the comparative measure (such as the max temperature a year ago). For more information, see the KendoReact Bullet Chart documentation.

This example React application generates a bullet chart using the TwoLayersChart widget:

```
import './App.css';
import { TwoLayersChart } from "ml-fasttrack";

function App() {

  const BulletChartConfig = {
    entityTypeConfig: {
        path: 'extracted.*~'
    },
    entities: [
        {
            entityType: 'person',
            items: [
                {
                    maxValue: { path: 'extracted.person.range.to' },
                    minValue: { path: 'extracted.person.range.from' },
                },
            ],
        },
        {
            entityType: 'organization',
            items: [
```

```
                {
                    maxValue: { path: 'extracted.organization.range.to' },
                    minValue: { path: 'extracted.organization.range.from' },
                },
            ],
        }
    ]
  }

  return (
    <div className="App">
      <div style={{width: '480px'}}>
        <TwoLayersChart
          data={exampleData}
          config={BulletChartConfig}
          chartType="bullet"
          settings={{
            bulletChartProps: {
              chartSeriesItemProps: {
                color: '#fff',
                data: [
                  [
                    52,
                    55
                  ]
                ]
              },
              chartSeriesLabelsProps: {
                font: 'bold 12pt sans-serif',
                position: 'center'
              },
              chartTitleProps: {
                text: 'Range'
              },
              chartTooltipProps: {
                render: () => {}
              },
              chartValueAxisItemProps: {
                max: 100,
                min: 0
              }
            }
          }}
          style={{ height: 120, width: 500 }}
          onPlotAreaClick={(e) => console.log("Clicked the plot area: ", e)}
          onSeriesClick={(e) => console.log("Clicked the series: ", e)}
        />
      </div>
    </div>
  );
}
```

```
export default App;
```

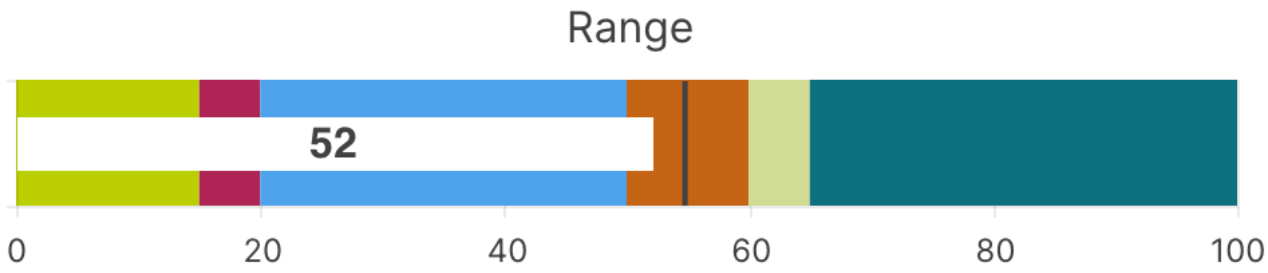
Code explanation

In the Display a bullet chart from search results example:

- The `data` prop receives the search results for the chart.
- The `config` prop receives a configuration object that determines the properties to include in the chart.
- The `chartType` prop specifies the chart type. In the Display a bullet chart from search results example, the chart is a bullet chart.
- The `settings` prop defines the KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type. For configuration details, see API.
- For more details about configuration, see the KendoReact Bullet Chart documentation.

Example rendering

The code in Display a bullet chart from search results renders this:



Display a polar chart from search results

The TwoLayersChart widget can display a polar chart. Polar charts are scatter charts that display two-dimensional data series in polar coordinates. For more information, see the KendoReact Polar Chart documentation.

This React application generates a polar chart with the TwoLayersChart widget:

```
import './App.css';
import { TwoLayersChart } from "ml-fasttrack";

function App() {

  const PolarChartConfig = {
    entityTypeConfig: {
```

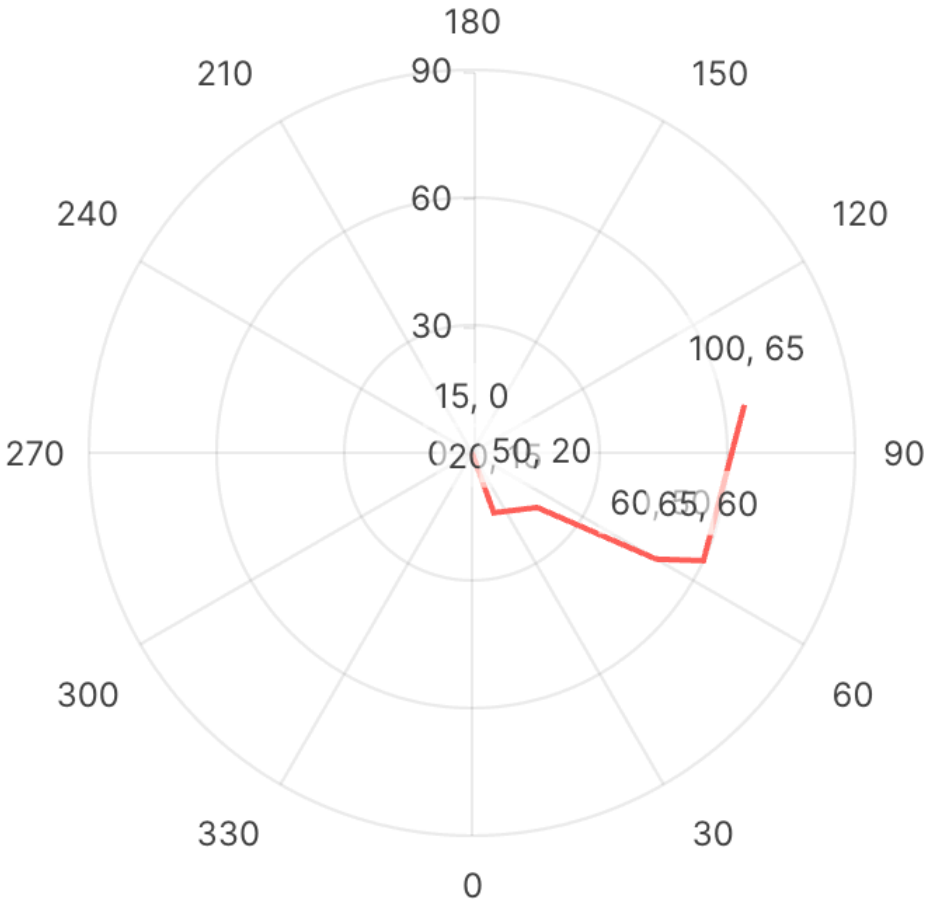
```
        path: 'extracted.*~'
    },
    entities: [
        {
            entityType: 'person',
            items: [
                {
                    maxValue: { path: 'extracted.person.range.to' },
                    minValue: { path: 'extracted.person.range.from' },
                },
            ],
        },
        {
            entityType: 'organization',
            items: [
                {
                    maxValue: { path: 'extracted.organization.range.to' },
                    minValue: { path: 'extracted.organization.range.from' },
                },
            ],
        }
    ]
}

return (
    <div className="App">
      <div style={{width: '480px'}}>
        <TwoLayersChart
          data={exampleData}
          chartType="polar"
          config={PolarChartConfig}
          settings={{
            polarChartProps: {
              chartProps: {
                  height: 500,
                  width: 500
              },
              chartseriesLabelProps: {
                  position: '',
                  content: ''
              },
              chartXAxisItemProps: {
                  startAngle: '-90',
                  majorUnit: '30'
              },
              chartYAxisItemProps: {
                  visible: false
              }
            }
          }}
          style={{ height: 500, width: 500 }}
          onPlotAreaClick={(e) => console.log("Clicked the plot area: ", e)}
```

```
          onSeriesClick={(e) => console.log("Clicked the series: ", e)}
        />
      </div>
    </div>
  );
}

export default App;
```

Code explanation

- The `data` prop receives the search results for the chart.
- The `config` prop receives a configuration object that determines the properties to include in the chart.
- The `chartType` prop specifies the chart type. In the Display a polar chart from search results example, the chart is a polar chart.
- The `settings` prop defines the KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type. For configuration details, see API.
- For more details about configuration, see the KendoReact Polar Chart documentation.

Example rendering



Display a scatterplot chart from formatted data

You can use the TwoLayersChart widget to display a scatterplot chart, which renders numerical data over two independent axes--X and Y. For more information, see the KendoReact Scatterplot Chart documentation.

The following React application generates a scatterplot chart with the TwoLayersChart widget. This example uses data that is already formatted for display by the widget. The widget expects an array of objects with `from` and `to` properties. Optionally, the `transformData` prop can be set to a transformation function and the data in the `data` prop will be transformed prior to being charted.

```
import './App.css';
import { TwoLayersChart } from "ml-fasttrack";

function App() {
```

```
  const exampleDataFormatted = [{ from: 5, to: 12 }, { from: 2, to: 6 }, {
from: 6, to: 9 }]

  return (
    <div className="App">
      <div style={{width: '480px'}}>
        <TwoLayersChart
          data={exampleDataFormatted}
          chartType="scatterPlot"
          settings={{
            scatterPlotChartProps: {
              chartTitleProps: {
                text: 'Scatter Example'
              }
            }
          }}
          style={{ height: 400, width: 400 }}
          onPlotAreaClick={(e) => console.log("Clicked the plot area: ", e)}
          onSeriesClick={(e) => console.log("Clicked the series: ", e)}
        />
      </div>
    </div>
  );
}

export default App;
```

Code explanation

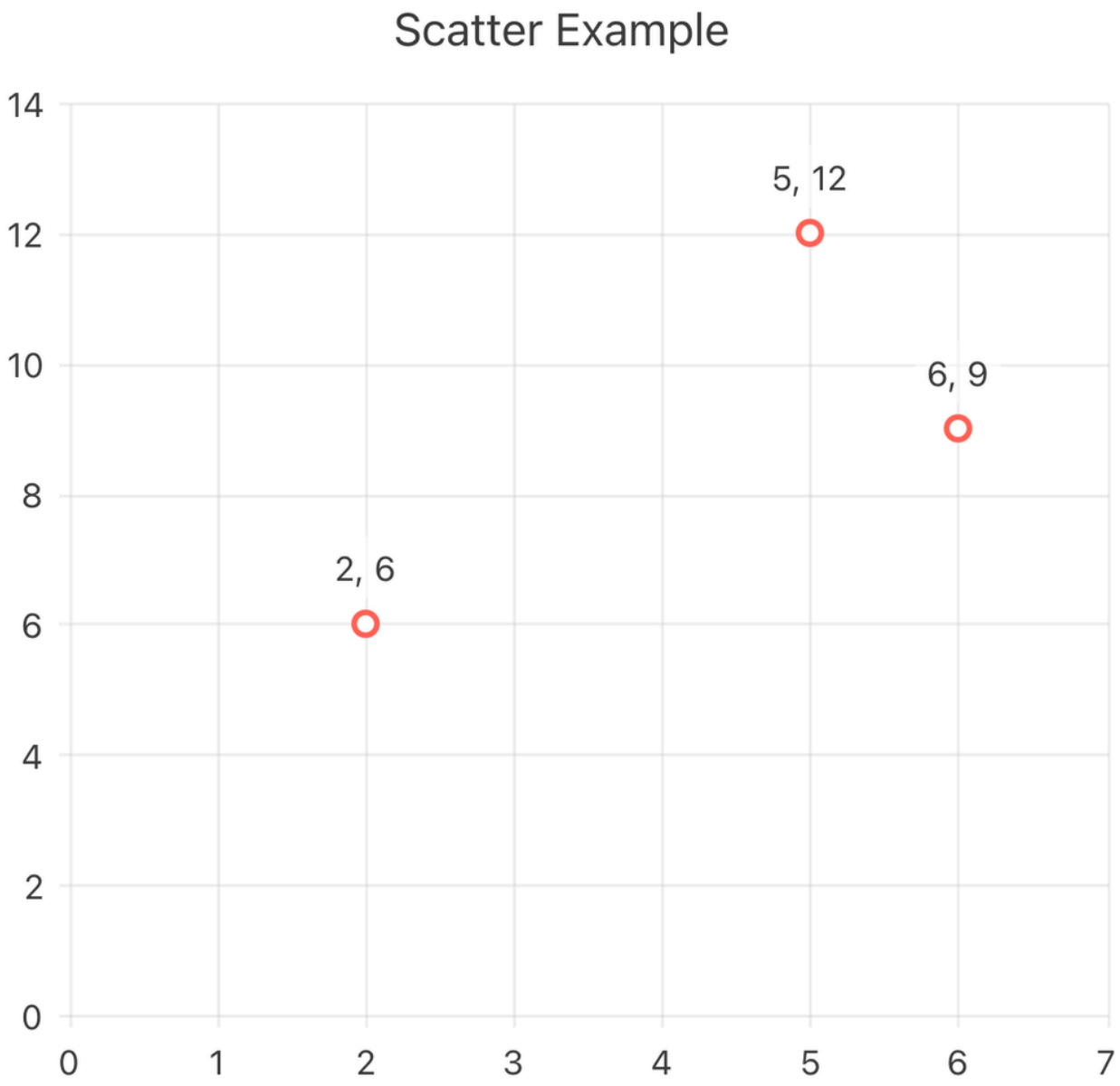
In the example:

- The information to chart is specified with the `data` prop.
- The `chartType` is set to display a scatterplot chart.
- The `settings` prop defines the KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type. For configuration details, see API.
- For more details about configuration, see the KendoReact Scatter Chart documentation.

> **Note:**
>
> For additional configuration information, see API, config API, and *ChartProps API.

Example rendering

The example code renders this:



API

Prop	Type	Description
data	object	Data to display in the chart.
config	object	Array of entity configuration objects. Each object determines what data to display in a result for an entity type. For additional details, See API, config API, and *ChartProps API.
chartType	"bullet" | "polar" | "scatterPlot"	The chart type to use in order to show data.
settings	object	KendoReact props passed in for configuring the chart. KendoReact props are passed in as objects corresponding to the chart type: • `bulletChartProps` • `polarChartProps` • `scatterPlotChartProps` For details, see the *ChartProps API table and the KendoReact Chart documentation.
onPlotAreaClick	function	Callback function triggered when the chart plot area is clicked. See KendoReact ChartProps.
onSeriesClick	function	Callback function triggered when the chart series is clicked. See KendoReact ChartProps.
style	CSSProperties	CSS styles applied to the chart.

Prop	Type	Description
transformData	function	Callback function for transforming data value. `transformData` retrieves a data object as an argument and returns an array of objects with `from` and `to` properties. The `transfromData` function cannot be used if a `config` prop is also used.

config API

Prop	Type	Description
entityTypeConfig	PathConfig	Entity type configuration object.
entityTypeConfig.path	string	Path to the entity type in the search result. The path is specified using JSONPath.
entities[]	object[]	Array of chart configuration objects for each entity.
entities[].entityType	string	Entity type of the configuration object.
entities[].items[]	object[]	An array of item configuration objects for the data to be charted. An array allows you to chart multiple property values in the data.
entities[].items[].minValue	PathConfig	Minimum value configuration object.
entities[].items[].minValue.path	string	JSONPath to minimum value.

Prop	Type	Description
entities[].items[].maxValue	PathConfig	Maximum value configuration object.
entities[].items[].maxValue.path	string	JSONPath to maximum value.
facet	object	Facet configuration object.
facet.name	string	Name of the facet whose values you are displaying in the chart.
facet.title	string	Title of the facet. This value is required for setting facet constraints in the application context.

*ChartProps API

Prop	Type	Description
chartProps	object	ChartProps for the KendoReact Chart component.
chartTitleProps	object	ChartTitleProps for the KendoReact Chart component.
chartSeriesProps	object	ChartSeriesProps for the KendoReact Chart component.
chartSeriesItemProps	object	ChartSeriesItemProps for the KendoReact Chart component.
chartSeriesLabelsProps	object	ChartSeriesLabelsProps for the KendoReact Chart component.
chartValueAxisProps	object	ChartValueAxisProps for the KendoReact Chart component.

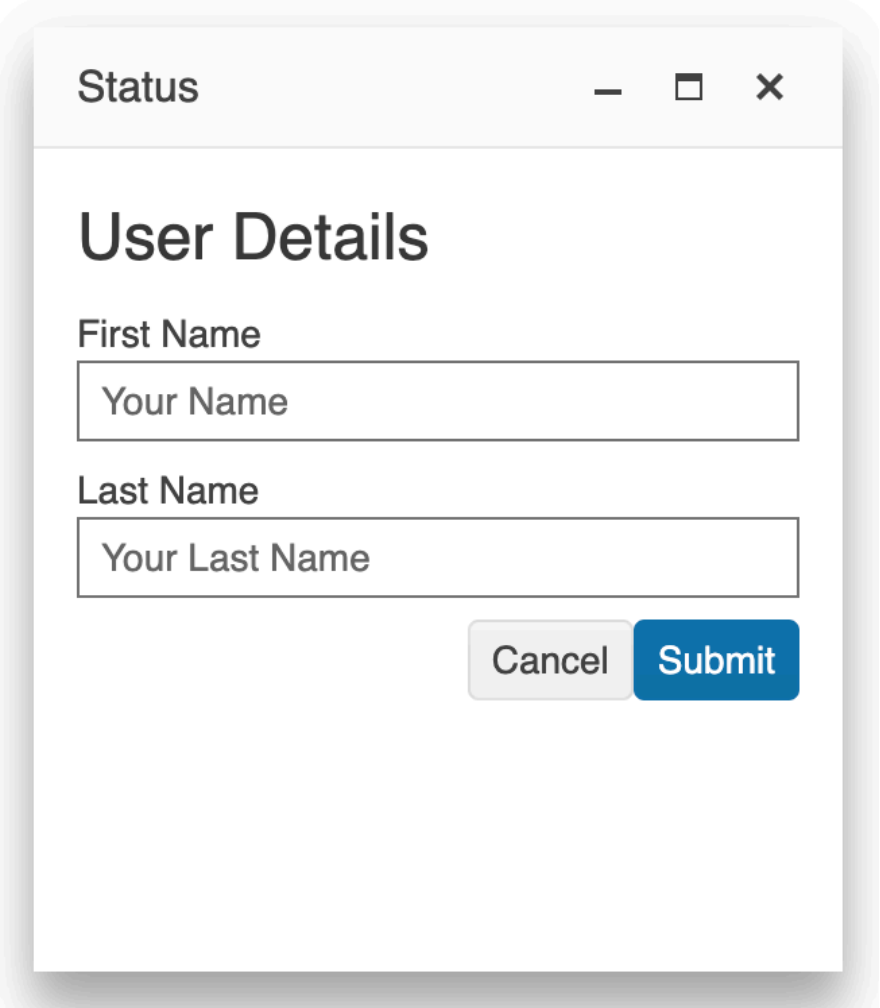
Prop	Type	Description
chartValueAxisItemProps	object	ChartValueAxisItemProps for the KendoReact Chart component.
chartXAxisProps	object	ChartXAxisProps for the KendoReact Chart component.
chartXAxisItemProps	object	ChartXAxisItemProps for the KendoReact Chart component.
chartYAxisProps	object	ChartYAxisProps for the KendoReact Chart component.
chartYAxisItemProps	object	ChartYAxisItemProps for the KendoReact Chart component.
chartTooltipProps	object	ChartTooltipProps for the KendoReact Chart component.
chartSeriesItemTooltipProps	object	ChartSeriesItemTooltipProps for the KendoReact Chart component.

WindowCard

The WindowCard widget displays content inside a popover window. The window can be moved, resized, or closed. The widget can enclose the EntityRecord widget to display formatted data from a document.

WindowCard example rendering

This example shows the WindowCard widget displaying content from a form.



WindowCard example configuration

In this example, the WindowCard widget displays information in a popover container. The string value in the `title` prop is the title displayed in the header of the modal. The `visible` prop accepts a boolean value that determines when the WindowCard is rendered on the page. Additional props can control the WindowCard's sizing and open/minimize/close behavior. See WindowCard API.

```
import { useContext } from "react";
import './App.css';
import { MarkLogicContext, WindowCard } from "ml-fasttrack";

function App() {

  return (
    <div className="App">
      <div ref={myRef}>
        <WindowCard
            title={"Status"}
```

```
            draggable={false}
            appendTo={myRef.current}
            visible={true}
            initialLeft={dimensions.width - 465}
        >
        {"User Details"}
        </WindowCard>
    </div>
  );
}

export default App;
```

WindowCard API

Prop	Type	Description
title	ReactNode	Title of the window.
appendTo	null | HTMLElement	Defines the container to which the WindowCard will be appended. Defaults to the parent element. See the KendoReact WindowProps.
toggleDialog	function	Callback that fires when the close button in the title is clicked or the ESC key is pressed.
visible	boolean	Visibility of the window.
draggable	boolean	Specifies if the window will be draggable. See the KendoReact WindowProps.
resizable	boolean	Specifies if the window will be resizable. See the KendoReact WindowProps.

Prop	Type	Description
initialWidth	number	Specifies the initial width of the window. See the KendoReact WindowProps.
initialHeight	number	Specifies the initial height of the window. See the KendoReact WindowProps.
initialTop	number	Specifies the initial top value of the window. See the KendoReact WindowProps.
initialLeft	number	Specifies the initial left value of the window. See the KendoReact WindowProps.
width	number	Specifies the width of the window. See the KendoReact WindowProps.
height	number	Specifies the height of the window. See the KendoReact WindowProps.
children	ReactNode	Content wrapped by the WindowCard widget displayed in the window.
stage	"DEFAULT" | "FULLSCREEN" | "MINIMIZED"	Controls the state of the window. See the KendoReact WindowProps.
maximizeButton	React.ComponentType<any>	Specifies if the window will render the maximize button. See the KendoReact WindowProps.
minimizeButton	React.ComponentType<any>	Specifies if the window will render the

Prop	Type	Description
		minimize button. See the KendoReact WindowProps.
restoreButton	React.ComponentType<any>	Specifies if the window will render the restore button. See the KendoReact WindowProps.
closeButton	React.ComponentType<any>	Specifies if the window will render the close button. See the KendoReact WindowProps.
onStageChange	(() => void)	Callback function triggered when the state of the window changes. See the KendoReact WindowProps.