Vision docs

Table of contents:

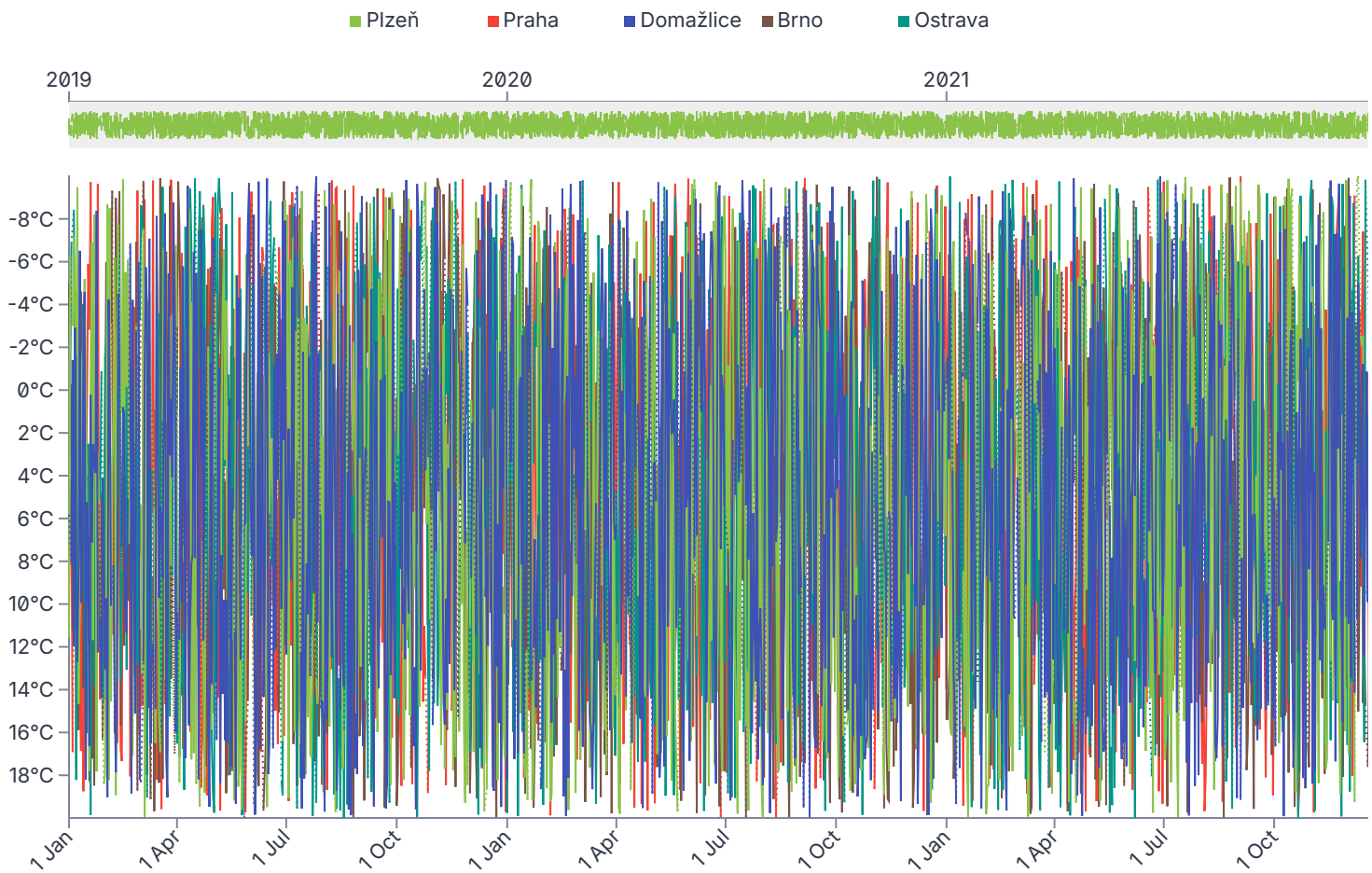
Vision

Build interactive charts and other visualizations using primitives in a single JSON object.

BEFORE USED

Please keep in mind that Vision is currently in **alpha stage**.

If you find a bug, please report to @Michal Kacerovsky. Thanks.



Basic Concepts

What's Different

- There is only one general component that **does not** specify what the visualization will look like.
- All visual and data input specifics are described in a single strictly validated JSON object called Vision specification (or shortly *spec*)
- Thanks to the specification, the visualization can be easily serialized, stored in a database and transferred between different applications.
- Input data does not need to be pre-transformed. Vision does not limit what data are on the input. If needed, it transforms the data itself based on the instructions in the JSON object.

Vision Props

Other visualization libraries usually expect a set of various parameters, such as axis description, coloring information or formatting, and accepts a specific data format only. Vision requires just three things – specification, input data, and context.

Data input

Vision accepts data in any format and provide them when defining datasets, offering the means to transform them. When using the data input, you're expected to define a necessary chain of transformations (a pipeline) resulting in a flat data set structure. For instance, the following input

```
[
    {
        "time": "2021-01-01",
        "temperatures": { "Pilsen": 3, "Prague": 1 }
    },
    {
        "time": "2021-01-02",
        "temperatures": { "Pilsen": -2, "Prague": 0 }
    }
]
```

should be modified by transformations to the following flat structure:

```
[
    {"time": "2021-01-01", "city": "Plzeň", "temperature": 3}
    {"time": "2021-01-01", "city": "Prague", "temperature": 1}
    {"time": "2021-01-02", "city": "Plzeň", "temperature": -2}
    {"time": "2021-01-02", "city": "Prague", "temperature": 0}
]
```

The resulting format is the basis for any further visualization processing.

> **TIP**
>
> Providing input data is not necessary. If you want to just play with Vision features, any of built-in example data sources will do just fine. Learn more at Data Pipelines.

Context

Context provides the visualization with additional functionality/information indirectly associated with the displayed data. This can be, for example, listeners for events fired by the visualization or default properties of certain visualization entities.

Specification

All the content of the visualization from the processing of input data to the rendering of visual elements is described by the specification – a JSON object whose structure is covered in detail by the specification schema. In addition, it is also validated before its processsing which makes most of the errors detected before the rendering of the visualization itself. Sections of chapter Specification Reference focus on individual parts of the JSON.

Terminology

You will often come across the terms listed in this table.

Term	
visualization	any visual representation of data
chart	a visual representation where data are represented by graphical elements such as rectangles or lines
specification	a JSON object completely describing the visualization
input	any kind of data passed to Vision
datum	a single data record formed as an object literal, e.g. `{time: "2020-01-01", value: 12.3}`
dataset	a flat list of items (datum), e.g. `[{time: "2020-01-01", value: 12.3}, {time: "2020-01-02", value: -2.5}]`
(data) pipeline	a named dataset created from data of a specified format and a set of transformations
scale	an abstraction that projects data dimension to a visual representation, e.g. a value to a vertical position or a color in a specific range
mark	a visual representation of data, e.g. rectangle, circle, text, line.
single-datum mark	a mark that can be created based on a single datum , e.g. reactangle or circle
multi-datum mark	a mark that that requires set of datums, usually consists of multiple points, e.g. line or area

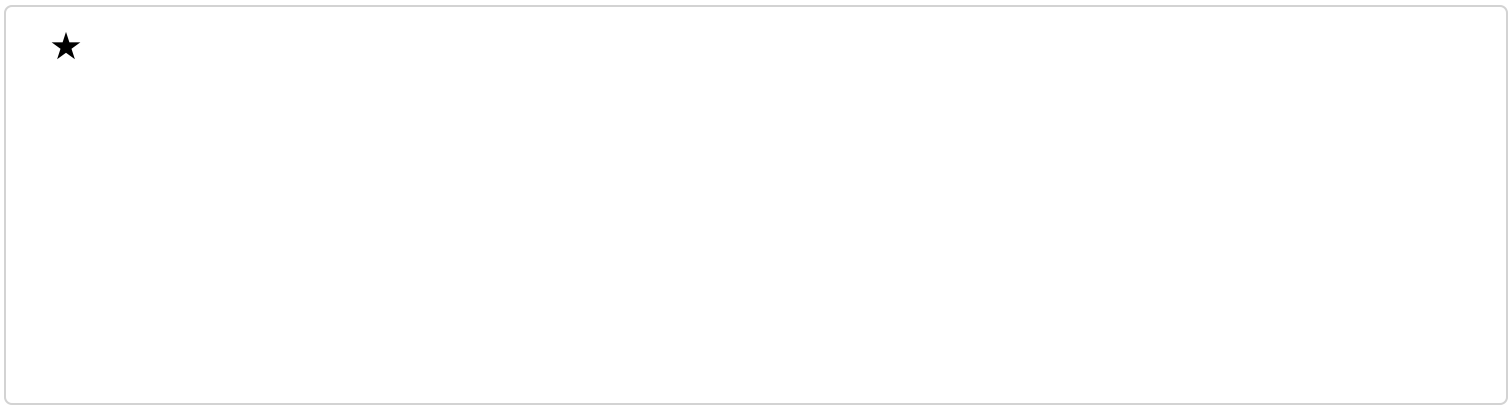
Term	
encoding	an object literal defining how to map data values to graphical properties using scales, expressions, formatting or explicit values
signal	a reactive variable whose value can updated with events in the visualization
paint	any kind of fill (color, pattern, gradient)

Usage

Minimal Example

```
import { Vision, VisionContextProvider } from '@sbks/vision'
import { createRoot } from 'react-dom/client'
const container = createRoot(document.getElementById('app'))

const spec = {
    marks: [
        {
            type: 'point',
            encode: {
                glyph: 'star',
                fill: '#000',
                x: 30,
                y: 20,
            },
        },
    ],
}
container.render(
    <VisionContextProvider>
        <Vision spec={spec} />
    </VisionContextProvider>
)
```

The example above doesn't use any data, just renders a point:



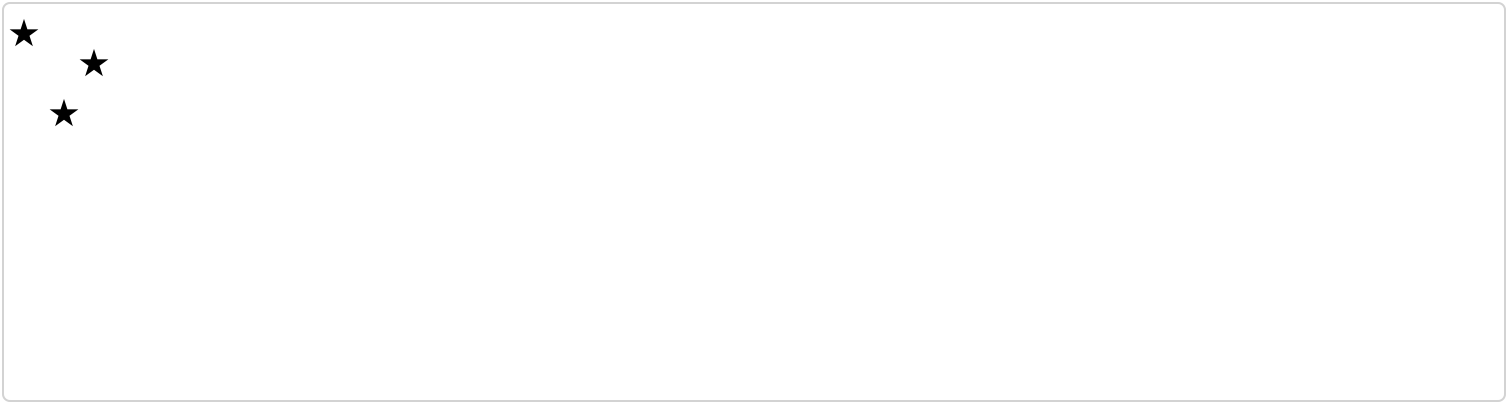
Now let's bring data into play. Say that the input for Vision will be a set of coordinates of the points we want to plot:

```
const DATA = [
    { x: 10, y: 15 },
    { x: 30, y: 55 },
    { x: 45, y: 30 },
]
```

The input data `DATA` is passed to the visualization as the `input` property. Passing the input itself will not change anything until we define a pipeline in the spec. In the example below, no `name` is specified for the pipeline, so Vision treats it as `root`.

Another change from the previous example is the connection of mark to data (`source`). This causes mark to be rendered for each dataset record instead of a single instance, providing the datum values within the encoding.

```
import { Vision, VisionContextProvider } from '@sbks/vision'
import { createRoot } from 'react-dom/client'
const container = createRoot(document.getElementById('app'))

const spec = {
    data: [{ source: '$input' }],
    marks: [
        {
            type: 'point',
            source: 'root',
            encode: {
                glyph: 'star',
                fill: '#000',
                x: { field: 'x' },
                y: { field: 'y' },
            },
        },
    ],
}
container.render(
    <VisionContextProvider>
        <Vision spec={spec} input={DATA} />
    </VisionContextProvider>
)
```



Theming

Let's stick with the previous example, but suppose that instead of a single set of coordinates, we want to display multiple sets - each item of `DATA` as a separate visualization.

```
const DATA = [
    [
        { x: 10, y: 15 },
        { x: 30, y: 55 },
        { x: 45, y: 30 },
    ],
    [
        { x: 40, y: 45 },
        { x: 10, y: 35 },
        { x: 25, y: 20 },
    ],
    [
        { x: 30, y: 45 },
        { x: 30, y: 35 },
        { x: 15, y: 10 },
    ],
]
```

Although we will be rendering three different datasets, the visualization is the same in principle, and therefore its spec can be used for all instances of Vision. We will use a similar spec as in the previous examples:

```
const SPEC = {
    data: [{ source: '$input' }],
    marks: [
```

```
        {
            type: 'point',
            source: 'root',
            encode: {
                glyph: 'star',
                x: { field: 'x' },
                y: { field: 'y' },
            },
        },
    ],
}
```

Within the Vision context we render three `Vision` components, each with different input data but the same spec.

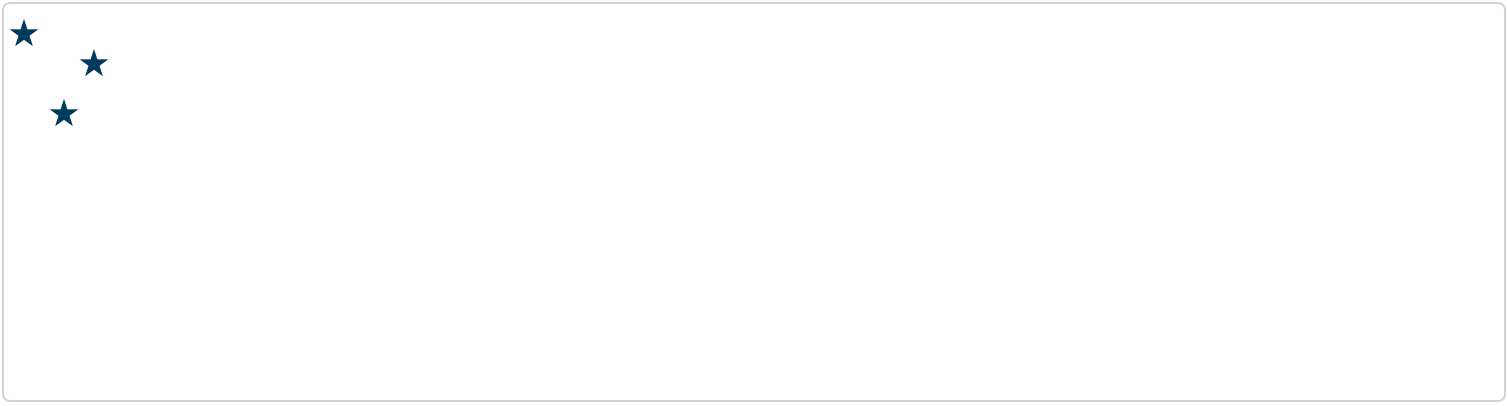
```
import { Vision, VisionContextProvider } from '@sbks/vision'
import { createRoot } from 'react-dom/client'
const container = createRoot(document.getElementById('app'))

container.render(
    <VisionContextProvider>
        <Vision spec={SPEC} input={DATA[0]} />
        <Vision spec={SPEC} input={DATA[1]} />
        <Vision spec={SPEC} input={DATA[2]} />
    </VisionContextProvider>
)
```

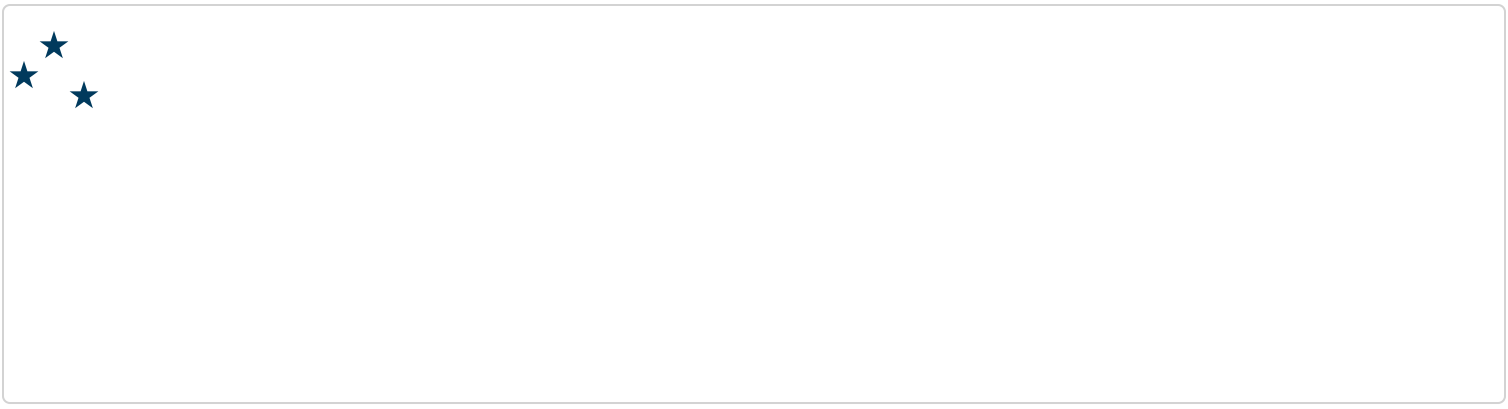
Vision does not render anything yet because the spec does not include any fill color for the point mark. Instead, we'll specify point fill within a theme, which makes it the default value for all Vision instances in the same context.

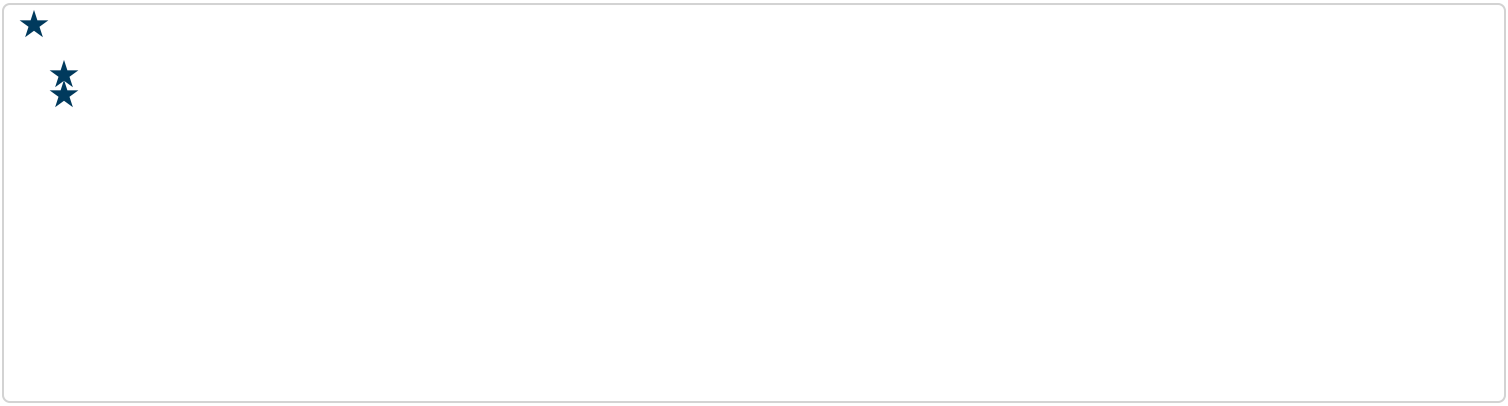
```
import { createVisionTheme } from '@sbks/vision'
const theme = createVisionTheme({
    defaults: { point: { encode: { fill: '#003A5D' } } },
})
```

Then pass it as prop to the context:

```
<VisionContextProvider theme={theme}>
```







Now all Vision instances within this context share the same default point mark color. If we want to prevent this behavior, we just need to explicitly specify the fill color in the spec.

> **INFO**
>
> See Theming to learn more about changing mark defaults, integrating Inter font and to inspect the default theme.

Development

1. Clone the repo.
2. Use the Node version matching *.nvmrc*.
3. Install NPM dependencies.
4. Run `npm run dev` – watches *src* folder and serves Vision playground app at http://localhost.aws.ccl:3001.

Publishing

> **NOTE**
>
> Before publishing, don't forget to create a migration script in *migration* folder for all changes that can be automatically migrated.

1. Run `npm version <semver>`.
2. Push the created tag `git push origin <tag>`.

Documentation

1. Init Docusaurus project by running `npm run docs:init`.
2. Run `npm run docs:dev`.

> **TIP**
>
> NPM script `docs:init` automatically installs the latest released version of Vision when initializing a Docusaurus project to be able to display interactive examples. However, if you are updating docs before releasing a new version, you probably need to use the unreleased version in the docs. The `docs:install-lib` script replaces the installed version with a freshly built and packaged library without its publishing.
>
> To restore the latest version, just run `docs:init` script again.

NPM scripts

NPM script	
`docs:init`	Initiates Docusaurus project in the *pages* folder and installs additional dependencies.
`docs:sync`	Copies *docs* folder and LCOV coverage stats to the Docusaurus project.
`docs:sync`	Copies *docs* folder, LCOV coverage stats, built playground and changelog to the Docusaurus project. (usually not necessary for dev)
`docs:dev`	Runs Docusaurus project at http://localhost:3000/vision
`docs:build`	Builds Docusaurus project.
`docs:clean`	Removes Docusaurus project (the *pages* folder).
`docs:install-lib`	Replaces Vision package used by Docusaurus project by a freshly built and packed library without its publishing.
`build`	Builds Vision lib and its validation schema.
`playground:build`	Builds the playground app to be integrated in docs.
`migration:init`	Creates a new migration file for the unreleased version
`migration:release`	Links the unreleased migration to the to-be-released version and update migration index (runs as a part of `npm version`)
`dev`	Watches *src* folder and serves Vision playground app at http://localhost:3001.
`lint`	Runs Eslint.
`test:coverage`	Runs Mocha tests with coverage.

NPM script	
`test`	Runs Mocha.

Conventions

Filenames

- Use kebab-case for all filenames.
- Use index files for exports only.
- Add an extension to the specific type of file as follows:

Pattern	
**.test.js*	Mocha test files
**.parse.js*	Contains related parsing functions
**.runtime.js*	Contains related functions for runtime evaluation
**.const.js*	Contains related constants
**.util.js*	Contains related utility functions
**.schema.js*	Exports related JSON schema

Specification fields

- Use camelCase for naming spec fields.
- Prefix mark state with colon, e.g. `:hover`.
- When allowing encodings other than root, use colon as separator, e.g. `encode:grid` or `encode:label`.
- Follow SVG attribute names where relevant.

How It Works

The JSON specification builds on the cross-reference mechanism – almost any part of the object can refer to instances created in another part. For example, a rectangle's position depends on a scale instance whose range is based on the minimum and maximum values in the data. Similarly in the same visualization, there may be another scale that also depends on the extent of the same data. In order to avoid repetitive computing of the same values (the data extent in this case), and to clarify what depends on each other, the specification undergoes parsing.

Parser stores a list of nodes where each node corresponds to an object that may carry a value - it may be a scale, a mark, an axis, but also an expression or a mark encoding. Besides, it creates a list of node references for all referenceable instances such as scales or data pipelines. E.g. once a scale definition appears in the specification, a node is created, added to the node list and also to the scale list. Then, if there is an element in the specification that refers to the defined scale, the name of the scale in parameters of the element's node is replaced by a reference obtained from the scale list.

When a parsed specification is ready, runtime starts processing with the starting points being a list of axes, legends, and the root mark. Runtime is gradually going through the dependencies between each node evaluating their value that is cached. Certain nodes of "value" type that depend on a dataset field or another node of this type are not cached. Their value is recalculated for each datum.

Runtime transforms the mark nodes into an object that contains, in addition to the mark type as stated in the original specification, the type of React component related to the mark type and the props to be passed.

Mark React components are usually simple functional components rendering SVG elements based on the passed paramaters almost identical to fields of mark encoding.

For common mark types, one visual instance, such as a rectangle, arc or text corresponds to each datum, and so each one is encoded and renderered individually. By contrast, line or area marks render whole dataset as a single visual instance. For such a component, all visual characteristics are identical for each datum.

Step by Step

Initiating the Runtime

Runtime is the core of visualization. It maintains the parsed specification and the values of each its part, and when any change occurs, it triggers an update of these values across all the dependencies that the specification creates.

When an instance of `Vision` component is created (and so `useVision` hook is created), first of all, the passed specification is validated and if is valid, `Runtime` is instantiated with passed data input, container element, value of `VisionContext`, event subscriber and any event handlers provided as `on*` props.

Then, `Runtime#init` initiates `Parser` and requests parsing the specification.

Initiating the Parser

The parser inits the *node list* `_nodes` that keeps all parsed nodes at index corresponding to their ID and the node registry `_registry` – a map that holds references to the parsed nodes of a specific type, such as *scale*, *axis* or *dataset*. Then, the parse process begins.

Parsing the Specification

Each specific node type has its own parsing function (stored in **.parse.js* source files), the parser just keeps track of the references and creates the nodes. These functions transform the specification into a uniform form and create dependencies. Their output is a list of parameters, which is then part of the resulting node.

These parameters do not have to be in a flat structure. If nested, the only rule is that the leaves are literals or references to other nodes `{$ref: <node ID>}`.

`Parser#parse` goes through these steps:

1. Adds nodes for built-in signals.
2. Registers scales. Scales can depend on each other, therefore they are registered only at first to be seamlessly referenced, and parsed later.
3. Parses custom signals.
4. Parses formats.
5. Parses data pipelines.
6. Parses scales.

7. Parses axes.
8. Parses legends.
9. Creates a spec for a group mark (the *root mark*) that contains all the marks from the spec and parses it.
10. Inits resolving references.

Resolving References

To resolve references, the parser needs some entry points at which should start – these are

- all legends,
- all axes,
- and the root mark.

The parser goes through these entry points and checks if any of their params contain a reference (`{$ref}` object literal). If so, it

- gets the referenced node by the ID specified in `$ref`
- adds ID of the referencing node to the *referenced-by* list (`refs`) of the referenced node,
- adds ID of the referenced node to the *dependent-on* list (`deps`) of the referencing node,
- and continues with resolving params of the referenced node.

As seen, all references are traversed in this way until eventually a dependency tree is created for each entry point, usually ending with nodes of type *encoding*, *value* or *signal*.

This ends the parser's work and the processed list of nodes is passed to the runtime together with the node registry.

Preparing Parsed Nodes for Runtime

The runtime extends nodes received from the parser with the additional fields:

- `value` that holds result of node evaluation (initiated to `undefined`),
- `outdated` that indicates that the value of the node is not up to date.

Besides these two extra fields, each node already includes:

- `id` used for referencing the node and specifying node's index in the node list,

- `type` determining how the node should be evaluated,
- `params`
- `deps` listing IDs of nodes on that is dependent,
- `refs` listing IDs of nodes that references the node (are dependent on)

Initiating Built-in Signals

Since the runtime has access to the container element and also information about whether any axes or legends are present in the visualization, it can determine values of the built-in signals.

In addition, it can also activate the `ResizeObserver`, which responds to any change in the dimensions of the container element by requesting an update of the built-in signal values.

Runtime Run!

At this point, the visualization is ready to be evaluated. All the steps that have taken place up to this point **will not need to be repeated** until the specification changes.

Just like parsing started with entry points, the runtime starts evaluating individual nodes from the same nodes, i.e. legends, axes and the root mark.

The main role here plays the `_evaluateNodeValue` function, which decides which evaluation function should be used to get the node value based on the node type. The evaluation functions can be found in the **.runtime.js* source files and evaluate the node params. If a dependency (a reference to another node) is encountered, they request the runtime to evaluate it. The runtime thus evaluates all referenced nodes in turn and stores the calculated value in their `value` field.

The result is the evaluated legend, axis and the root mark configurations. These can be passed to the `Vision` component.

Rendering Runtime Output

Before running the runtime, the `useVision` hook attached *render* event handler to the runtime. This event is fired each time any of the entry points is re-evaluated and passes the configurations for rendering. `useVision` stores these configurations in a state which leads to re-rendering of the `Vision` component.

`Vision` just iterates the configurations and use corresponding components to render it. Voilà, the visualization is rendered.

Handling the Interactivity

But Vision is not static. All the following events trigger a runtime update - each in a specific way.

The Input Changed

If the input data change and the specification includes a pipeline that depends on it, the runtime starts re-evaluating the values for these pipelines. Thus, only those nodes that depend on the dataset are affected.

The Container Dimension Changed

If the ResizeObserver registers a change in the dimensions of the container element, triggers an update of the built-in signals.

A Custom Signal Value Changed

This is the only case where the change is requested directly by the author of the specification. If an event is fired and is expected to update the signal value according to the specification, the runtime updates the signal node value.

Re-evaluation

All of the above cases are handled by updating the node value. Since these nodes are of *signal* or *dataset* type, other nodes are dependent on them, and so their value must also be recalculated.

1. The runtime traverses the dependecy tree in which the to-be-updated node appears and flags each dependent node as *outdated*.
2. The new value is set to the to-be-updated node and `Runtime#run` called.
3. `Runtime#run` evaluates the nodes the same way as in the first run but when encouters a node that is not flagged as *outdated*, uses the cached value available in its `value` field. Otherwise, computes new value and toggles the *outdated* flag.

Migrate to 1.0.0

This document only lists breaking changes or recommendations to replace some notation with newly implemented ones. To discover what's new, see API Reference or changelog.

- 🚨 Breaking change
- ⚠ Deprecated / Behavior change
- ℹ️ Recommendation

Migration Script

To automate the migration of existing legacy specifications, the `migrate` function included in the package can be used. For upgrading the spec from 0.1.11 to the current version, call `migrate("0.1.11", "current", spec)`. The migration function covers as many breakpoint changes as possible, but some cannot be migrated automatically:

- MathJS expressions used in `expr` fields and in `predicate` of the mapping transformation are not migrated to JS automatically.
- Axes placed at the same orientation won't stack automatically (need `offset` specified in its encoding)

Package Exports

- 🚨 Vision JSON schema is now exported as `schema` (not `visionSpecSchema`).

Schema

Values

[Full Reference]

- ℹ️ `invert` field is now available, so `{expr: "invert('<scale>', 10)"}` can be replaced by `{invert: "<scale>", value: 10}` which is preferred.

- 🚨 `expr` changed from MathJS to a JS expression or set of JS statements
  - MathJS function are not supported, but JS globals such as `Math` or `Array` are.
  - Indexing arrays changed (MathJS uses 1-based indexing).

Data Pipelines

[Full Reference]

- 🚨 `source` is **required**.
- 🚨 `source` does not support index access.
- 🚨 All test sources are all available as `"$test"`, particular source is accessed via new property `path`
- 🚨 Dataset transformations distinguish `field` (intented for single field) and `fields` (intented for multiple fields)

Scales

[Full Reference]

- 🚨 `domain` and `range` are **required**.
- 🚨 `domain.min` (resp. `domain.max`) moved to `domainMin` (resp. `domainMax`)

Axes

[Full Reference]

- 🚨 `caption` is `string | Signal<string>`, caption encoding (former `caption.encode`) is changed to the encoding target `encode:caption`.
- 🚨 `size` is of `number` type only.
- 🚨 `offset`, `stroke` and `strokeWidth` is part of `encode`.
- 🚨 `label.format` moved to `format`.
- 🚨 `label.encode` changed to encoding target `encode:label`.
- 🚨 `label.render` moved to the encoding target `encode:label` as `encode:label.render`
- 🚨 `tick.count` moved to `ticks`

Legends

[Full Reference]

- 🚨 `type` is **deprecated** since it's derived from type of `scale`.
- 🚨 `size` is of `number` type only.
- 🚨 `symbol` changed to the encoding target `encode:symbol`

Marks

[Full Reference]

- 🚨 `encode.tooltip` moved to the root and is expected to be `boolean | Signal<boolean>`.
- 🚨 `encode.interactive` moved to the root and is expected to be `string | Signal<string>`.
- 🚨 `encode:hover` changed to an encoding state, moved to `encode[":hover"]`.

Behaviour

Marks

- ⚠ **Text**: `align` is only effective if `width` specified, otherwise the text orientation is affected by `anchorX`