Plume

Project documentation
unreleased

Contents

Run an example 3
Working with charts 4
- Construct a figure 4
- Coordinates and styling 4
- Titles, legends and hover text 5
- Render the result 6
- Browse the API reference 6

API guarantees 7
- Keep the contract executable 8

Contribute to Plume 9
- Documentation 9
- Development and validation 9
- Release operation 10

Plume

Plume is a pure Roc package that builds Plotly chart specifications and complete HTML documents. It supports bar, scatter, pie, donut and Sankey charts. The package performs no I/O; applications choose where to send its output.

Run an example

Install the latest Roc nightly. From the repository root:

```
roc examples/simple.roc > chart.html
```

Open the resulting file in a browser. HTML loads Plotly from `cdn.plot.ly` with a fixed versioned URL and integrity hash; viewing the chart requires network access to that CDN. Chart labels and hover text follow Plotly's own text and markup rules. HTML document titles and embedded JSON are escaped separately.

Read Working with charts for constructors, styling and output, and API guarantees for validation and encoding contracts. The API reference describes individual public types and methods.

Working with charts

A chart has three layers: validated domain values, a trace describing one series, and a figure that combines traces with a layout. Applications import the modules they use from their `plume` dependency and decide where to write the resulting text.

Construct a figure

Inside an application's `main!`, create a trace and wrap it in the matching `Chart.Trace` tag:

```
sales = Bar.new({
    data: [
        { x: "Apples", y: 12 },
        { x: "Oranges", y: 8 },
    ],
    name: "Sales",
})?

chart = Chart.new({
    traces: [Bar(sales)],
    layout: Layout.new({ title: Title.new({ text: "Fruit sales" }) }),
})

html = chart.to_html()
```

`Chart.new` and `Layout.new` are infallible. Constructors that validate raw numeric settings, such as `Bar.new`, return `Try`; use `?` to propagate their structured errors. Empty datasets and an empty trace list are valid. Bar width uses axis units and may exceed one. A figure can contain several trace kinds, and each trace retains the order of its input points or slices.

Constructor	Use	Important settings
`Bar.new`	Vertical or horizontal bars	Per-point color, orientation, width in axis units
`Scatter.new`	Lines, scatter plots, bubbles and text	Mode, per-point marker and text, shared line style
`Pie.new`	Pies and donuts	Labeled nonnegative slices; hole in the closed interval zero to one
`Sankey.new`	Directed flows	Stable node IDs, links, hover overrides and orientation

Coordinates and styling

Cartesian point fields use `Coordinate`. Plain string and number literals convert at compile time; runtime data uses explicit constructors:

```
label = Coordinate.from_text(product_name)
quantity = Coordinate.from_number(measured_quantity)?
sales = Bar.new({ data: [{ x: label, y: quantity }] })?
```

Text coordinates remain text even when they contain digits. Numeric coordinates use F64 precision, and `Coordinate.from_number` rejects NaN and both infinities with `NonFiniteCoordinate(number)`. For date axes, supply appropriate date text and select `Axis`'s `Date` type; Plotly interprets the strings. Plume does not parse date syntax.

Markers and line styles are validated before they enter a scatter trace:

```
purple = Color.rgb({ red: 124, green: 56, blue: 245 })
marker = Marker.new({ size: 12, symbol: Diamond, color: purple })?
line = Line.new({ width: 2, color: purple, dash: Dash })?
points = Scatter.new({
    data: [{ x: 1, y: 2, marker }],
    mode: LinesMarkers,
    line,
})
```

`Scatter.new` is infallible because its coordinates and styles have already been checked. The `mode` selects lines, markers, text, or a combination; per-point `text` is drawn only by a mode that includes text. Point order is also line-connection order. A marker's size is its diameter in pixels, and a line's width is measured in pixels. Their zero sizes hide the corresponding shape; negative and nonfinite values are errors.

Use named symbol tags in source. `Marker.Symbol.from_str` accepts canonical Plotly names when reading configuration, returning `InvalidSymbol(text)` for unknown or unsupported names. `Color.hex` accepts six hexadecimal digits following `#`; `Color.named` accepts supported CSS color names without regard to ASCII case. RGB and alpha channels use bytes, so alpha zero is transparent and alpha 255 is opaque.

`Font.default`, `Axis.default` and `Title.default` leave their settings to Plotly. Calling their constructors applies the documented props explicitly. A `Font`'s size must be finite and at least one pixel. Font families are CSS family lists resolved by the browser; the package does not load fonts. Numeric axis bounds may run in either direction. On logarithmic axes the bounds are base-10 exponents: zero to two displays values from one to 100.

Titles, legends and hover text

`Layout`'s `title` is the title drawn in the chart. Its legend is hidden by default; set `legend: Shown` to display trace names. `Chart`'s separate `title` controls the HTML document title: `FromLayout` derives it from the visible title, with a fallback when no title is set; `Text(text)` overrides the document title without changing the chart or its JSON figure.

Sankey nodes use distinct string IDs independently of display labels. Links must reference existing, different IDs; duplicate IDs, absent endpoints and self-links produce separate errors from `Sankey.new`. Nonnegative finite flow values, including zero, are accepted. Cycles through distinct nodes are supported.

A node or link's `hover: Default` preserves Plotly's normal hover content. `Text(text)` overrides only that node or link. Chart labels and hover text follow Plotly's supported formatting rules. The package escapes their JSON representation; this does not turn Plotly's text formatting into plain text.

Render the result

`chart.to_json()` returns a complete figure object containing `data` and `layout`. Use it when another application manages Plotly and its document. `chart.to_html()` returns a complete HTML document with a pinned Plotly CDN script and an integrity hash. Viewing that document requires network access to the CDN. HTML document-title text and JSON embedded in the script element are escaped for their separate contexts.

Low-level `to_json_value` methods are composition hooks used by the package's renderers. Their `InternalJson` result is an opaque, safely encoded value; it has no unchecked raw-JSON constructor and is intentionally absent from the package's exposed modules. Applications normally use the complete `to_json` or `to_html` output instead of these hooks.

Browse the API reference

Browse the API reference. To generate it locally, run `blueprint run api-docs` and open `generated-docs/index.html`. The generated pages describe each constructor, its props, defaults, closed choices and failure tags. `blueprint run policy` checks that public definitions have attached documentation, and `blueprint run check` runs the same policy gate. Presence checks do not judge clarity or completeness: review the rendered pages whenever public types, options or behavior change.

API guarantees

Plume's API should make invalid chart states difficult to express and impossible to carry past a validated constructor. This is a product contract, enforced by types and executable tests rather than naming conventions alone.

Boundary	Guarantee	Enforcement
Cartesian coordinate	A category/date string or a finite F64 number	Opaque `Coordinate`; string/ number literal hooks; checked runtime numeric constructor
Style value	Valid size, width, color and closed symbol/dash options	Opaque types, props and structured constructor errors
Trace	Validated points and presentation; aligned arrays preserve point order	Non-generic opaque trace types and decoded-output tests
Sankey graph	Unique IDs, existing endpoints, nonnegative finite flows and no self-links	Constructor validation with distinct error tags
JSON	Only properly encoded scalars and structured arrays/objects	Opaque internal builder; no unchecked string-fragment constructor
HTML	Chart text cannot terminate its JSON script element or document title	Separate context escaping and adversarial

Boundary	Guarantee	Enforcement
		round-trip tests

`Coordinate.from_quote` and `Coordinate.from_numeral` let normal literals use the same checked boundary without wrapping every point. Runtime strings use `Coordinate.from_text`; runtime numbers use `Coordinate.from_number` and return `Try`. Numeric coordinates have F64 precision. Records, lists and booleans are not coordinates. Date strings are interpreted by Plotly according to the axis configuration; the coordinate type does not validate a date grammar.

Constructors return errors only when their input can be invalid. `Scatter.new` is infallible because its coordinates, marker and line already carry their guarantees. `Bar.new`, `Pie.new` and `Sankey.new` validate their additional numeric or graph inputs. Rendering a valid value needs no new user-input validation. Text resembling template placeholders remains literal data in the encoded output.

Keep the contract executable

`blueprint run check` runs four complementary forms of evidence:

- Named unit tests check exact output, enum mappings and boundary failures.
- Consumer tests use the public API and independently decode the JSON.
- Type-contract tests compile valid controls and require invalid programs to fail for the intended reason, including opacity bypass attempts.
- The roc-fuzz target generates text, floating-point boundaries and graph cases, checking constructor policy and safe populated-chart round trips.

Type-contract fixtures are data; the driver writes temporary Roc applications and requires a normal compiler rejection with the expected diagnostic. A warning, timeout, signal or unrelated setup failure cannot satisfy a case. The corresponding positive control must compile successfully first.

The fuzz smoke checks the target's declared examples and committed allocation baseline before a bounded exploration. The check task owns the build options, execution budget and process deadline. Its output identifies skipped checks and records diagnostics for failures.

Review changes to expected output and allocation baselines as code changes. Use the test reports to understand the observations behind a baseline change; routine checks never accept new expectations automatically.

When changing a public type, demonstrate both a valid intended use and the invalid construction it excludes. When a runtime constructor gains a rule, add its boundary cases and extend the relevant generated-input property. Keep error tags distinct and avoid adding a fallible return to an operation whose input types already guarantee success.

Contribute to Plume

Documentation

Every paragraph should help someone use, maintain or assess the current project. Delete superseded instructions and duplicate explanations. Keep implementation history in Git, unresolved work in issues, and release-specific changes on the published release; do not include release notes in the manual. Preserve current compatibility guidance, security boundaries, API contracts and required notices.

Use the shared documentation guide for structure and publishing.

Development and validation

Use the Blueprint release declared in `Blueprint.roc` and the latest Roc nightly as `roc` on `PATH`:

```
blueprint check
blueprint run check
blueprint run docs
blueprint run release-package -- build 0.0.1-dryrun
blueprint run release-test -- test-archive
```

The documentation build includes the manual and generated API reference in `.docs-out/site/`. Pull requests retain that site for preview; the default branch deploys the same build to GitHub Pages.

`Blueprint.lock` records the digest and length of each released utility and the fixed compiler used for repository scripts. The product compiler is supplied by the host and by CI. Checks use the dev backend. These are correctness checks, not performance measurements.

`check` formats sources, runs package and consumer `expect` tests, and runs every example against `tests/spec.json`. Expected output changes are recorded with `blueprint run release-test -- test-examples --update` and reviewed as a diff. The release utility builds each example once and compares output, stderr and exit status for each case. Failed commands and timeouts fail the run. The release archive is tested separately against the exact bundled package over loopback.

The README gallery uses the same images as the visual CI check. With Node available, run `blueprint run screenshots-setup` to install the configured browser, then `blueprint run screenshots` to rebuild the examples and compare fresh renders with those images. A mismatch retains expected, actual and difference images for review. For an intentional visual change, update any changed example output first, then run `blueprint run screenshots-update` and review the PNG diff before committing.

The workflows pin shared roc-automation code and use its pure-package model. Roc source follows the style guide.

Release operation

Pull requests run the shared release dry run. Publication requires dispatching `Release` from the default branch with a version, after reviewing its artifacts and configuring the repository's `release` environment. The required checks are `Release / Build and test the source` and `Release / Test the release candidate`, plus the documentation job if required by repository policy. Adding these workflow files does not configure GitHub branch rules or environment protection.

The nightly workflow checks the latest published release, rather than the working tree. It requires a published examples archive that follows the shared release contract. Local checks and dry runs do not establish hosted CI or publication success.