**1. General Layout and Padding**

json

Copy

"autosize": "pad",

"padding": {

"left": 5,

"right": 0,

"top": 5,

"bottom": 0

}

* **autosize**: This property ensures that the visualization resizes according to the container it’s placed in (in this case, Power BI). The "pad" value ensures there’s padding around the visualization.
* **padding**: This section specifies the space (in pixels) between the edges of the chart and the content inside. You've set small padding on the left and top, but none on the right and bottom.

**2. Signals**

Signals in Vega are dynamic variables that can be updated based on interaction or data. They control aspects like sizing, color, and layout.

json

Copy

"signals": [

{

"name": "height",

"update": "pbiContainerHeight-65"

},

{

"name": "width",

"update": "pbiContainerWidth"

},

{

"name": "showTooltips",

"value": true

},

{

"name": "showButtons",

"value": true

},

{

"name": "showDomainSpanLabel",

"value": false

},

{

"name": "textColour",

"value": "Black"

},

{

"name": "coloursDark",

"value": [

"#000080",

"#3d7922",

"#808080"

]

},

{

"name": "coloursLight",

"value": [

"#000080",

"#3d7922",

"#808080"

]

}

]

* **height** and **width**: These signals dynamically calculate the height and width of the visualization based on the container’s size. This ensures responsiveness to different Power BI report sizes.
* **showTooltips**, **showButtons**, **showDomainSpanLabel**: These signals control visibility. For example, you may want to toggle tooltips or buttons on/off or hide the span labels for specific tasks.
* **textColour**: Sets the color of text throughout the chart, in this case, black.
* **coloursDark** and **coloursLight**: These signals define color schemes for your chart. These are arrays with color values that you can use for task bars, labels, or other elements.

**3. Row and Column Layouts**

json

Copy

{

"name": "yRowHeight",

"value": 33,

"description": "Height in pixels"

},

{

"name": "portfolioColumnWidth",

"value": 120

},

{

"name": "pmwColumnWidth",

"value": 100

},

{

"name": "programColumnWidth",

"value": 250

},

{

"name": "taskColumnWidth",

"value": 155

}

* **yRowHeight**: This signal controls the height of each row. You set it to 33 pixels, determining how tall each task bar or phase bar will appear.
* **portfolioColumnWidth**, **pmwColumnWidth**, **programColumnWidth**, etc.: These signals define the widths of various columns in your chart (such as the portfolio column, program column, etc.). By defining these, you're able to create a structured layout that accommodates the necessary information for each task.

**4. Time and Date Settings**

json

Copy

{

"name": "oneDay",

"update": "1000\*60\*60\*24"

},

{

"name": "dayBandwidth",

"update": "scale('x', timeOffset('day', datetime(2000,1,1),1)) - scale('x', datetime(2000,1,1))"

},

{

"name": "today",

"update": "utc(year(now()),month(now()),date(now()))"

}

* **oneDay**: The number 1000 \* 60 \* 60 \* 24 represents the number of milliseconds in one day. This helps define time intervals in the Gantt chart.
* **dayBandwidth**: This defines the width of one day on the chart. It uses the scale function to calculate the spacing between each day based on the current time scale.
* **today**: This signal uses the utc() function to store today’s date in UTC format. This will be used to draw a vertical line or marker representing the current day.

**5. Layout and Data Widths**

json

Copy

{

"name": "columnsWidth",

"update": "taskColumnWidth + startColumnWidth + endColumnWidth + portfolioColumnWidth + pmwColumnWidth + programColumnWidth + (columnPadding \* 8)"

},

{

"name": "ganttWidth",

"update": "width - columnsWidth - minDayBandwidth"

}

* **columnsWidth**: This calculates the total width of the non-Gantt chart columns. It takes into account the widths of the various information columns such as task name, start date, and portfolio column, plus padding.
* **ganttWidth**: This calculates the remaining width for the Gantt chart itself, excluding column widths and padding. It's essentially the space where tasks will be drawn along the timeline.

**6. Dynamic Time Extent and Gantt Chart Width**

json

Copy

{

"name": "xExt",

"update": "[data('xExt')[0]['s']-oneDay\*200, data('xExt')[0]['s'] + ((ganttWidth - minYearBandwidth)/minYearBandwidth)\*.9\*oneDay]"

}

* **xExt**: This signal defines the time range for the Gantt chart's X-axis. It starts 200 days before the earliest start date (s) and extends to a calculated point based on the chart's width. This ensures the chart’s timeline fits within the available space.

**How the Code Works in the Visual:**

1. **Responsiveness**: The chart adjusts its size dynamically based on the Power BI container size. This is achieved by the height and width signals.
2. **Columns and Layout**: You’ve set specific widths for the columns (e.g., task column, start date column, etc.), and the chart will adjust according to these predefined values. The layout signals ensure a consistent, structured appearance for the Gantt chart with appropriate padding.
3. **Time Representation**: You’ve defined how time is represented on the X-axis using the oneDay, dayBandwidth, and xExt signals. The width of a day is calculated dynamically based on the available space.
4. **Colors and Tooltips**: The visual provides flexibility in customization, including different color schemes (light/dark) and enabling tooltips to show task details on hover.
5. **Today’s Marker**: A "Today" line (based on the today signal) can be displayed on the chart to show the current date, which is useful in a Gantt chart to track progress.

This additional section of your Vega code introduces dynamic behaviors related to zoom, panning, item interaction, and layout adjustment. It allows for user interactivity, such as zooming in/out of the timeline and panning through the chart. Here's a breakdown of the key components:

**1. Zooming Logic**

{

"name": "zoom",

"value": 1,

"on": [

{

"events": "wheel!",

"force": true,

"update": "x() > columnsWidth ? pow(1.001, (event.deltaY) \* pow(16, event.deltaMode)) : 1"

}

]

}

* **zoom**: This signal listens for wheel events (mouse wheel or trackpad pinch gestures).
* **update**: The zoom value is adjusted based on the wheel movement. The event.deltaY value is used to determine the zoom factor, with a slightly exponential scaling applied via pow(). If the mouse is outside the column area (x() > columnsWidth), zooming will happen on the Gantt chart timeline.

**2. Dynamic Time Span (xDom)**

{

"name": "xDomMinSpan",

"update": "span(xExt)"

},

{

"name": "xDomMaxSpan",

"update": "round((ganttWidth / 0.13) \* oneDay)"

},

{

"name": "xDom",

"update": "xExt",

"on": [

{

"events": {

"signal": "xDomPre"

},

"update": "span(xDomPre) < xDomMinSpan ? [anchor + (xDom[0] - anchor) \* (zoom \* (xDomMinSpan / span(xDomPre))), anchor + (xDom[1] - anchor) \* (zoom \* (xDomMinSpan / span(xDomPre)))] : span(xDomPre) > xDomMaxSpan ? [anchor + (xDom[0] - anchor) \* (zoom \* (xDomMaxSpan / span(xDomPre))), anchor + (xDom[1] - anchor) \* (zoom \* (xDomMaxSpan / span(xDomPre)))] : xDomPre"

},

{

"events": { "signal": "delta" },

"update": "[xCur[0] + span(xCur) \* delta[0] / width, xCur[1] + span(xCur) \* delta[0] / width]"

}

]

}

* **xDomMinSpan** and **xDomMaxSpan**: These signals define the minimum and maximum span (range) of the timeline in terms of days. xDomMinSpan ensures the timeline has a minimum time range, while xDomMaxSpan ensures it doesn't exceed a maximum range based on the Gantt chart's width.
* **xDom**: This signal governs the X-axis range. It dynamically adjusts based on zoom or user interaction, ensuring the zoomed-in area or panned region stays within the valid time range (between xDomMinSpan and xDomMaxSpan). If the user zooms or pans, it updates accordingly.

**3. Panning and Dragging (yRange)**

{

"name": "yRange",

"update": "[yRange != null ? yRange[0] : 0, yRange != null ? yRange[0] + scaledHeight : scaledHeight]",

"on": [

{

"events": [{ "signal": "delta" }],

"update": "clampRange([yCur[0] + span(yCur) \* delta[1] / scaledHeight, yCur[1] + span(yCur) \* delta[1] / scaledHeight], height >= scaledHeight ? 0 : height - scaledHeight, height >= scaledHeight ? height : scaledHeight)"

}

]

}

* **yRange**: This signal controls the vertical (Y-axis) range. It adjusts the visible area of the chart based on user input (such as panning). The scaledHeight represents the total height required to display all the rows of tasks.
* **update**: The clampRange() function ensures that the chart doesn’t scroll out of bounds vertically. The range is adjusted dynamically based on how much the user has panned up or down (via delta[1]).

**4. Anchoring and Cursor Position**

{

"name": "anchor",

"value": 0,

"on": [

{

"events": "wheel",

"update": "+invert('x', x() - columnsWidth)"

}

]

},

{

"name": "xCur",

"value": [0, 0],

"on": [

{

"events": "pointerdown",

"update": "slice(xDom)"

}

]

}

* **anchor**: This signal tracks the X position when the user interacts with the Gantt chart (e.g., when using the wheel to zoom). It is used to maintain the position of the zoom relative to the current visible area, ensuring the zoom or pan is focused around the point of interaction.
* **xCur**: This signal stores the current X-axis range when the user clicks or drags on the chart (pointerdown event). It enables panning or zooming based on the point of interaction.

**5. Item Hover and Click Events**

{

"name": "itemHovered",

"value": { "id": "", "dependencies": [] },

"on": [

{

"events": "@taskSelector:mouseover,@phaseOutline:mouseover,@taskBars:mouseover,@taskNames:mouseover,@taskLabels:mouseover",

"update": "{'id': toString(datum.id), 'dependencies':split(datum.dependencies,',')}"

},

{

"events": "@taskTooltips:mouseover",

"update": "{'id': toString(datum.datum.id), 'dependencies':split(datum.datum.dependencies,',')}"

},

{

"events": "@taskSelector:mouseout,@phaseOutline:mouseout,@taskBars:mouseout,@taskNames:mouseout,@taskLabels:mouseout,@taskTooltips:mouseout",

"update": "{'id': '', 'dependencies':[]}"

}

]

},

{

"name": "hover",

"value": "",

"on": [

{

"events": "@buttonMarks:pointerover",

"update": "datum.text?datum.text:''",

"force": true

},

{

"events": "@buttonMarks:pointerout",

"update": "''",

"force": true

}

]

}

* **itemHovered**: This signal captures which task or element the user is hovering over and updates the id and dependencies fields. This allows the chart to track task-related hover events (for example, displaying task details or showing a tooltip).
* **hover**: This is a helper signal that tracks which button is being hovered over. It updates the tooltip or hover label based on the button's text (such as "Open" or "Close").

**6. Closing and Opening Views**

{

"name": "closeAll",

"on": [

{

"events": "@buttonMarks:click",

"update": "datum.text == 'Close' ? true : false",

"force": true

}

]

},

{

"name": "openAll",

"on": [

{

"events": "@buttonMarks:click",

"update": "datum.text == 'Open' ? true : false",

"force": true

}

]

}

* **closeAll** and **openAll**: These signals handle button clicks for "Close" or "Open" actions, likely toggling between different zoom levels or showing/hiding parts of the chart (e.g., collapsing rows or expanding details).

**Summary of Interactivity**

This code adds powerful interactive features to your Gantt chart, such as:

* **Zoom**: Using the mouse wheel, users can zoom in and out on the timeline, adjusting the view based on the zoom signal.
* **Panning**: Users can drag the chart to scroll horizontally or vertically through the tasks using xCur, yCur, and related signals.
* **Hover and Click Events**: Hovering over tasks or phases triggers updates (e.g., showing tooltips), and clicking on tasks or buttons can update the view or trigger additional actions.
* **Button Marks**: Buttons like "Open" and "Close" can control the chart’s zoom level or toggle visibility of certain elements.

These interactivity features make your Gantt chart more engaging and user-friendly, allowing users to explore tasks, zoom in on specific timeframes, and navigate the data smoothly.

**1. Data Sources:**

"data": [

{"name": "dataset"},

{

"name": "collapsedphases",

"on": [

{

"trigger": "phaseClicked",

"toggle": "phaseClicked"

},

{

"trigger": "closeAll",

"remove": true

},

{

"trigger": "closeAll",

"insert": "data('phases')"

},

{

"trigger": "openAll",

"remove": true

}

]

},

...

]

* **dataset**: This is the main data source containing the tasks and project data. Each task has information such as start time, end time, dependencies, completion percentage, and related project information.
* **collapsedphases**: This tracks whether certain phases of the project are collapsed or expanded. It uses triggers such as phaseClicked, closeAll, and openAll to toggle the visibility of different phases of the Gantt chart:
  + **phaseClicked**: Toggles the phase visibility.
  + **closeAll**: Closes all the phases and removes them from the visualization.
  + **openAll**: Opens all collapsed phases.

**2. Transformations:**

Each of the datasets goes through different transformations to shape the data for visualization.

**input Transformation:**

This transformation applies several calculations to derive fields used in the Gantt chart:

{

"name": "input",

"source": "dataset",

"transform": [

{

"type": "formula",

"as": "start",

"expr": "utc(year(datum.start), month(datum.start), date(datum.start))"

},

...

]

}

* **start and end**: Converts the start and end date values into proper DateTime objects (utc function is used here to ensure that the dates are interpreted in UTC format).
* **days**: Calculates the number of days between the start and end dates by subtracting the start from the end and dividing by the oneDay constant (defined earlier).
* **portfolio, program, currenttask**: These fields are just passed from the original dataset to be used for filtering and grouping.

**phases Transformation:**

This transformation aggregates the data by phase and performs additional calculations:

{

"name": "phases",

"source": "input",

"transform": [

{

"type": "aggregate",

"fields": ["start", "end", "completion", "task", ...],

"ops": ["min", "max", "sum", "count", "mean", ...],

"as": ["start", "end", "sum", "count", "completion", ...],

"groupby": ["phase"]

},

...

]

}

* **aggregate**: The tasks are grouped by phase, and multiple aggregation functions (min, max, sum, count) are used:
  + **start**: The minimum start date of all tasks in the phase.
  + **end**: The maximum end date of all tasks in the phase.
  + **completion**: The sum or mean of the task completion percentages.
  + **task**: Grouped by the phase, representing the specific tasks under that phase.
* This allows for creating a summary view of each phase, such as the total duration and completion status of all tasks within the phase.

**phasePaths Transformation:**

This transformation defines the graphical representation of each phase, specifically the paths for each phase on the chart:

{

"name": "phasePaths",

"source": "phases",

"transform": [

{

"type": "formula",

"as": "phasePath",

"expr": "'M ' + scale('x', datum.start) + ' ' + (scale('y', datum.id) + yPaddingInner) + ' H ' + scale('x', datum.end) + ' ' + ' v ' + phaseSymbolHeight + ' L ' + (scale('x', datum.end) - phaseSymbolWidth) + ' ' + (scale('y', datum.id) + yPaddingInner + phaseSymbolHeight / 2) + ' L ' + (scale('x', datum.start) + phaseSymbolWidth) + ' ' + (scale('y', datum.id) + yPaddingInner + phaseSymbolHeight / 2) + ' L ' + (scale('x', datum.start)) + ' ' + (scale('y', datum.id) + yPaddingInner + phaseSymbolHeight) + ' z'"

}

]

}

* **phasePath**: A formula calculates the path for each phase, which will be used to render the shape (likely a rectangle or polygon) that represents the phase on the timeline. The M (move to) and L (line to) commands are used to define the coordinates of the phase's start and end.
  + The scale('x', datum.start) and scale('x', datum.end) determine the position on the X-axis (time).
  + The scale('y', datum.id) determines the position on the Y-axis, and padding values (like yPaddingInner) adjust the spacing.
  + phaseSymbolHeight and phaseSymbolWidth define the dimensions of the graphical phase symbol.

**3. Signals and Interactivity:**

These signals are used for interactive actions like zooming, panning, and expanding/collapsing the phases.

{

"name": "zoom",

"value": 1,

"on": [

{

"events": "wheel!",

"force": true,

"update": "x()>columnsWidth?pow(1.001, (event.deltaY) \* pow(16, event.deltaMode)):1"

}

]

},

...

* **zoom**: Handles zooming functionality based on mouse wheel input (wheel! event). The x() function is used to determine the current zoom level, and pow(1.001, event.deltaY) is used to scale the zoom factor when scrolling.

{

"name": "phaseClicked",

"value": null,

"on": [

{

"events": "@taskSelector:click,@phaseOutline:click",

"update": " yCur[0]==yRange[0] && yCur[1]==yRange[1] && xCur[0]===xDom[0] && xCur[1]===xDom[1] && datum.phase==datum.task? {phase: datum.phase}:null"

}

]

}

* **phaseClicked**: When a phase or task is clicked, this signal is triggered. It checks if the clicked phase corresponds to the current visible section (xCur, yCur, xDom, yRange) and updates the phase value.

{

"name": "closeAll",

"on": [

{

"events": "@buttonMarks:click",

"update": "datum.text=='Close'?true:false",

"force": true

}

]

}

* **closeAll**: This signal checks whether the user clicked a "Close" button (datum.text=='Close') and triggers the closing of all phases. The force: true ensures that the action is immediately applied.

**4. Final Steps and Other Sections:**

You will also see transformations that allow for the interaction with **tooltips**, **hover states**, and **task selections**.

* **hover and itemHovered**: These manage hover states when the user hovers over various parts of the chart (tasks, phases, labels, etc.). It keeps track of the hovered element's information and can be used for displaying tooltips or highlighting.
* **closeAll and openAll**: These interact with buttons to close or open all phases in the chart.

**How It All Works:**

1. **Data Preparation**: The raw dataset is transformed to calculate the start, end, and other properties (e.g., completion) for tasks, and to create aggregated data for phases. This allows the visualization to show phase-level aggregates and individual task timelines.
2. **Interactive Features**:
   * **Zoom**: The user can zoom in and out of the timeline using the mouse wheel. The zoom level is adjusted based on the delta of the wheel event.
   * **Pan/Scroll**: The user can scroll horizontally to move the visible window over the timeline and vertically to adjust which tasks or phases are visible.
   * **Phase Collapse/Expand**: Clicking on a phase or using buttons allows the user to collapse or expand phases in the chart.
3. **Rendering**: The chart is drawn using scales (like scale('x') and scale('y')) to position the phases and tasks. Paths are generated for phases, and tasks are represented visually based on their duration and position on the timeline.

**1. tasks Data Transformation:**

{

"name": "tasks",

"source": "input",

"transform": [

{

"type": "filter",

"expr": "!indata('collapsedphases', 'phase', datum.phase)"

}

]

}

* **Purpose**: This transformation filters out tasks based on whether their corresponding phase is collapsed.
* **Explanation**:
  + **indata('collapsedphases', 'phase', datum.phase)**: Checks if the current datum.phase exists in the "collapsedphases" dataset.
  + **!indata(...)**: The ! negates this check, meaning the filter will exclude tasks whose phases are listed in the "collapsedphases" dataset (i.e., tasks in collapsed phases are excluded).

**2. yScale Data Transformation:**

{

"name": "yScale",

"source": [

"tasks",

"phases"

],

"transform": [

{

"type": "lookup",

"from": "phases",

"key": "phase",

"values": ["phaseSort"],

"fields": ["phase"]

},

{

"type": "window",

"sort": {

"field": [

"phaseSort",

"taskSort"

],

"order": [

"ascending",

"ascending"

]

},

"ops": ["row\_number"],

"as": ["finalSort"]

}

]

}

* **Purpose**: This transformation creates a vertical scaling (yScale) for the tasks and phases.
* **Explanation**:
  1. **lookup**:
     + Looks up the phaseSort values from the phases dataset, based on the phase key.
     + This effectively associates each task with the sort order of its phase (sorting phases in order).
  2. **window**:
     + After the lookup, this operation applies a window function to sort tasks by their phaseSort and taskSort values (both in ascending order).
     + **row\_number** assigns a unique row number (finalSort) to each task, effectively determining the Y-position of each task in the visualization.

**3. xExt Data Transformation:**

{

"name": "xExt",

"source": "input",

"transform": [

{

"type": "aggregate",

"fields": ["start", "end"],

"ops": ["min", "max"],

"as": ["s", "e"]

},

{

"type": "formula",

"as": "days",

"expr": "(datum.e - datum.s) / oneDay"

}

]

}

* **Purpose**: This transformation calculates the minimum and maximum dates for tasks and computes the total duration in days.
* **Explanation**:
  1. **aggregate**:
     + **min** and **max** operations are applied to the start and end fields, respectively, calculating the earliest start time (s) and the latest end time (e) for the tasks.
  2. **formula**:
     + Computes the total duration of tasks (in days) by subtracting the start time from the end time and dividing by oneDay (a constant value representing a single day).
     + This field (days) will be used to determine the length of each task's bar in the Gantt chart.

**4. taskDependencyArrows Data Transformation:**

{

"name": "taskDependencyArrows",

"source": "yScale",

"transform": [

{

"type": "filter",

"expr": "isValid(datum.dependencies) && datum.dependencies!=''"

}

]

}

* **Purpose**: This transformation filters out tasks that do not have valid or defined dependencies.
* **Explanation**:
  + **isValid(datum.dependencies)**: Ensures that the dependencies field for each task is a valid value (i.e., not null or undefined).
  + **datum.dependencies != ''**: Ensures that the dependencies field is not an empty string.
  + Tasks that have no valid dependencies are excluded, and the remaining tasks with dependencies will be used to visualize dependency arrows between tasks.

**5. phaseDependencyArrows Data Transformation:**

{

"name": "phaseDependencyArrows",

"source": "input",

"transform": [

{

"type": "filter",

"expr": "indata('collapsedphases', 'phase', datum.phase)"

},

{

"type": "joinaggregate",

"fields": ["id", "start"],

"ops": ["values", "min"],

"as": [

"allphaseIds",

"start"

],

"groupby": ["phase"]

},

{

"type": "formula",

"as": "id",

"expr": "toString(datum.id)"

},

{

"type": "formula",

"as": "allphaseIds",

"expr": "pluck(datum.allphaseIds, 'id')"

},

{

"type": "formula",

"as": "dependencies",

"expr": "split(datum.dependencies, ',')"

},

{

"type": "flatten",

"fields": ["dependencies"]

},

{

"type": "formula",

"as": "internalDependenciesIndex",

"expr": "indexof(datum.allphaseIds, datum.dependencies)"

},

{

"type": "filter",

"expr": "datum.dependencies != 'null' && datum.dependencies != '' && datum.internalDependenciesIndex == -1"

},

{

"type": "lookup",

"from": "phases",

"key": "phase",

"values": ["id"],

"fields": ["phase"],

"as": ["id"]

}

]

}

* **Purpose**: This transformation processes phase dependencies and creates arrows between phases (e.g., showing dependencies between different phases).
* **Explanation**:
  1. **filter**: Filters phases that are collapsed (using indata('collapsedphases', 'phase', datum.phase)).
  2. **joinaggregate**:
     + This step aggregates the id and start fields by phase.
     + **values** returns all id values for the tasks in each phase.
     + **min** calculates the earliest start time for tasks within each phase.
  3. **formula**:
     + Converts id to a string format for easier use in further transformations.
     + **pluck(datum.allphaseIds, 'id')** extracts the id field from the array allphaseIds.
     + **split(datum.dependencies, ',')** splits the comma-separated list of dependencies into an array.
  4. **flatten**: Flattens the dependencies array so that each dependency is treated as a separate data entry.
  5. **indexof(datum.allphaseIds, datum.dependencies)**: Finds the index of the dependency in the allphaseIds list.
  6. **filter**: Filters out invalid dependencies (such as null or empty strings) and ensures that the dependency exists in the phase list (index of -1 means it doesn't exist in the allphaseIds).
  7. **lookup**: This performs a lookup on the phases dataset to retrieve the id for the phases with valid dependencies.

**Summary of the Code's Structure:**

* **tasks**: Filters out tasks that belong to collapsed phases.
* **yScale**: Creates the Y-axis scale for tasks, ensuring they are ordered correctly by phase and task order.
* **xExt**: Defines the overall project timeline by aggregating the earliest start time and latest end time.
* **taskDependencyArrows**: Filters and processes task dependencies for visualizing arrows between tasks.
* **phaseDependencyArrows**: Processes dependencies between phases, filtering out invalid ones and preparing data for dependency arrows between phases.

**1. dependencyArrows Data Transformation:**

{

"name": "dependencyArrows",

"source": [

"taskDependencyArrows",

"phaseDependencyArrows"

]

}

* **Purpose**: This step combines the results from taskDependencyArrows and phaseDependencyArrows.
* **Explanation**:
  + The dependencyArrows dataset is created by merging the two data sources: taskDependencyArrows (arrows between tasks) and phaseDependencyArrows (arrows between phases). This likely means that both task and phase dependencies will be visualized together in one set of arrows.

**2. dependencyLines Data Transformation:**

{

"name": "dependencyLines",

"source": [

"yScale",

"phaseDependencyArrows"

],

"transform": [

{

"type": "filter",

"expr": "isValid(datum.dependencies) && datum.dependencies != ''"

},

{

"type": "formula",

"as": "dependencies",

"expr": "split(datum.dependencies, ',')"

},

{

"type": "flatten",

"fields": ["dependencies"]

},

{

"type": "lookup",

"from": "input",

"key": "id",

"values": ["id", "end", "phase"],

"fields": ["dependencies"],

"as": ["sourceId", "sourceEnd", "sourcephase"]

},

{

"type": "lookup",

"from": "phases",

"key": "phase",

"values": ["id", "end"],

"fields": ["sourcephase"],

"as": ["sourcephaseId", "sourcephaseEnd"]

},

{

"type": "formula",

"as": "sourceId",

"expr": "indata('collapsedphases', 'phase', datum.sourcephase) == true ? datum.sourcephaseId : datum.sourceId"

},

{

"type": "formula",

"as": "sourceEnd",

"expr": "indata('collapsedphases', 'phase', datum.sourcephase) == true ? datum.sourcephaseEnd : datum.sourceEnd"

},

{

"type": "formula",

"as": "plottedStart",

"expr": "(scale('x', datum.start) + (dayBandwidth / 2) - sqrt(arrowSymbolSize)) - 1"

},

{

"type": "formula",

"as": "plottedSourceEnd",

"expr": "scale('x', datum.sourceEnd) - (dayBandwidth / 2)"

},

{

"type": "formula",

"as": "a",

"expr": "[scale('x', datum.start) + (dayBandwidth / 2), scale('y', datum.id) + bandwidth('y') / 2]"

},

{

"type": "formula",

"as": "b",

"expr": "[datum.plottedStart >= datum.plottedSourceEnd ? datum.plottedSourceEnd : datum.plottedStart, scale('y', datum.id) + bandwidth('y') / 2]"

},

{

"type": "formula",

"as": "c",

"expr": "[datum.plottedSourceEnd, scale('y', datum.sourceId) + bandwidth('y') / 2]"

},

{

"type": "formula",

"as": "d",

"expr": "[datum.plottedStart > datum.plottedSourceEnd ? null : datum.plottedStart, datum.plottedStart > datum.plottedSourceEnd ? null : scale('y', datum.sourceId) + bandwidth('y')]"

},

{

"type": "formula",

"as": "e",

"expr": "[datum.plottedStart > datum.plottedSourceEnd ? null : datum.plottedSourceEnd, datum.plottedStart > datum.plottedSourceEnd ? null : scale('y', datum.sourceId) + bandwidth('y')]"

},

{

"type": "fold",

"fields": ["a", "b", "d", "e", "c"]

},

{

"type": "filter",

"expr": "datum.value[0] != null"

},

{

"type": "formula",

"as": "value0",

"expr": "datum.value[0]"

},

{

"type": "formula",

"as": "value1",

"expr": "datum.value[1]"

},

{

"type": "window",

"ops": ["row\_number"],

"as": ["duplicates"],

"groupby": ["id", "sourceId", "value0", "value1"]

},

{

"type": "filter",

"expr": "datum.duplicates == 1"

}

]

}

* **Purpose**: This transformation processes and prepares the data for visualizing dependency lines (arrows) between tasks or phases.
* **Explanation**:
  1. **filter**: Excludes records where dependencies are either invalid or empty.
  2. **formula**:
     + Splits the dependencies field (which might be a comma-separated list) into individual dependencies.
     + The plottedStart, plottedSourceEnd, and other fields calculate positions for visualizing arrows between tasks or phases.
  3. **lookup**: Retrieves the id, end, and phase for each dependency from the input and phases datasets, based on the dependencies field.
  4. **formula**: Applies conditions to determine the correct start (sourceId) and end times (sourceEnd) for the dependencies, handling cases where phases are collapsed.
  5. **fold**: Transforms the multiple coordinates (a, b, c, etc.) into a pair of values (x, y) for each data point.
  6. **filter**: Filters out any entries where the coordinates are null.
  7. **window**: Identifies duplicate data points (likely for handling overlapping arrows) and assigns row numbers.
  8. **filter**: Removes duplicates, keeping only the unique dependency relationships.

**3. buttons Data:**

{

"name": "buttons",

"values": [

{

"side": "left",

"text": "Close",

"x": 15,

"leftRadius": 4

},

{

"side": "left",

"text": "Open",

"x": 65,

"rightRadius": 4

}

]

}

* **Purpose**: This section defines buttons (likely UI elements) for the visualization.
* **Explanation**:
  + Two buttons are created: one labeled "Close" and the other labeled "Open". Both buttons are positioned on the left side (side: "left") of the visualization.
  + The x position defines where each button is placed horizontally on the screen, and the leftRadius and rightRadius specify the rounded corner radius for the buttons.

**Summary:**

* **dependencyArrows**: Merges task and phase dependency data to visualize arrows representing the dependencies between tasks and phases.
* **dependencyLines**: Processes the dependency data and calculates positions and coordinates for arrows between tasks and phases. It handles edge cases, such as collapsed phases and overlapping dependencies.
* **buttons**: Defines two UI buttons ("Close" and "Open") with specified positioning and rounded corners for potential interactive controls within the chart.

This part of the code is focused on processing dependency relationships and creating interactive UI elements (like buttons) to control aspects of the visualization. The data transformations ensure that dependency arrows are correctly positioned and visually clear.

**1. buttonMarks:**

{

"name": "buttonMarks",

"description": "All buttons",

"type": "group",

"from": {"data": "buttons"},

"clip": {

"signal": "!showButtons"

},

"encode": {

"update": {

"x": {

"signal": "datum.side=='left'?datum.x:columnsWidth+ganttWidth-datum.x"

},

"width": {"value": 50},

"y": {"value": -60},

"height": {"signal": "18"},

"stroke": {

"signal": "'#7f7f7f'"

},

"strokeWidth": {"value": 1},

"cornerRadiusTopLeft": {

"field": "leftRadius"

},

"cornerRadiusBottomLeft": {

"field": "leftRadius"

},

"cornerRadiusTopRight": {

"field": "rightRadius"

},

"cornerRadiusBottomRight": {

"field": "rightRadius"

},

"cursor": {

"value": "pointer"

},

"fill": [

{

"test": "indexof(hover,datum.text)>-1",

"value": "#4e95d9"

},

{

"test": "datum.text=='Close' && data('collapsedphases').length == data('phases').length",

"value": "#4e95d9"

},

{

"test": "datum.text=='Open' && data('collapsedphases').length == 0",

"value": "#4e95d9"

},

{

"test": "datum.text=='Days' && dayBandwidthRound == minDayBandwidth",

"value": "#4e95d9"

},

{

"test": "datum.text=='Months' && dayBandwidthRound>=minYearBandwidth && dayBandwidthRound<minDayBandwidth",

"value": "#4e95d9"

},

{

"test": "datum.text=='Years' && dayBandwidthRound<minYearBandwidth",

"value": "#4e95d9"

},

{"value": "white"}

]

}

},

"marks": [

{

"name": "buttonText",

"interactive": false,

"type": "text",

"encode": {

"update": {

"text": {

"signal": "parent.text"

},

"baseline": {

"value": "middle"

},

"align": {

"value": "center"

},

"x": {

"signal": "item.mark.group.width/2"

},

"y": {"signal": "10"},

"fill": [

{

"test": "indexof(hover,parent.text)>-1",

"value": "white"

},

{

"test": "parent.text=='Close' && data('collapsedphases').length == data('phases').length",

"value": "white"

},

{

"test": "parent.text=='Open' && data('collapsedphases').length == 0",

"value": "white"

},

{

"test": "parent.text=='Days' && dayBandwidthRound == minDayBandwidth",

"value": "white"

},

{

"test": "parent.text=='Months' && dayBandwidthRound>=minYearBandwidth && dayBandwidthRound<minDayBandwidth",

"value": "white"

},

{

"test": "parent.text=='Years' && dayBandwidthRound<minYearBandwidth",

"value": "white"

},

{"value": "#7f7f7f"}

]

}

}

}

]

}

**Explanation:**

* **name**: The group of all button elements is named buttonMarks.
* **from**: The data for these buttons is sourced from the buttons dataset defined earlier.
* **clip**: The visibility of the buttons is controlled by the showButtons signal. If showButtons is false, the buttons are not rendered.
* **encode (update)**:
  + **x**: The horizontal positioning (x axis) of the button depends on whether the button is on the left or right side. If the side is "left", it uses datum.x for positioning; otherwise, it positions it relative to the combined width of columnsWidth and ganttWidth.
  + **width**: Sets a fixed width of 50 units for each button.
  + **y**: The vertical position of the buttons is set to -60 units (this may place the buttons above the main content).
  + **height**: The height is fixed at 18 units.
  + **stroke**: Sets a gray stroke color (#7f7f7f) for the button border.
  + **strokeWidth**: Specifies the width of the stroke as 1 unit.
  + **cornerRadius**: Defines the rounded corners of the button using leftRadius and rightRadius values.
  + **cursor**: When hovering over the button, the cursor is set to a pointer ('pointer'), indicating interactivity.
  + **fill**: The button's background color changes based on various conditions:
    - If the button is hovered (indexof(hover, datum.text) > -1), the fill color changes to a blue shade (#4e95d9).
    - Other conditions check the state of collapsed phases or specific text labels ("Close", "Open", "Days", etc.) and set the background color to blue if the conditions are met.
    - Default background color is white.
* **marks (button text)**:
  + A child mark of type text is used to render the button's text.
  + The text is aligned centrally within the button.
  + **fill**: The text color changes based on conditions similar to the button's background color (e.g., white when hovered or based on phase conditions).

**2. xDomainText:**

{

"name": "xDomainText",

"interactive": false,

"type": "text",

"encode": {

"update": {

"text": {

"signal": "showDomainSpanLabel ? timeFormat(xDom[0], '%d/%m/%y') + ' - ' + timeFormat(xDom[1], '%d/%m/%y') : null"

},

"baseline": {"value": "top"},

"align": {"value": "right"},

"x": {

"signal": "columnsWidth + ganttWidth"

},

"y": {

"signal": "showDomainSpanLabel ? height + 15 : 0"

},

"fill": {

"signal": "textColour"

}

}

}

}

**Explanation:**

* **name**: This mark represents the label for the domain (likely a date range) and is named xDomainText.
* **interactive**: Set to false, meaning the text is not interactive and will not respond to user input.
* **encode (update)**:
  + **text**: The text displayed is based on the xDom signal (likely a range of dates). If showDomainSpanLabel is true, it formats and displays the date range (xDom[0] to xDom[1]). If showDomainSpanLabel is false, the text is null, meaning no text is shown.
  + **baseline**: The vertical alignment of the text is set to the "top" of the position.
  + **align**: The horizontal alignment of the text is set to "right".
  + **x**: The horizontal position is set to the sum of columnsWidth and ganttWidth (positions it at the right side of the chart).
  + **y**: The vertical position depends on whether showDomainSpanLabel is true. If true, it places the label 15 units below the height of the visualization. Otherwise, the vertical position is 0 (no label).
  + **fill**: The text color is determined by the textColour signal, allowing dynamic changes to the text color.

**Summary of Key Features:**

* **buttonMarks**: These represent interactive buttons that can trigger actions such as "Open", "Close", or changing the display unit (e.g., "Days", "Months", "Years").
  + Buttons have dynamic styling based on user interaction (hover) and the state of collapsed phases.
  + The button text adjusts its appearance depending on conditions such as hovering or state changes.
* **xDomainText**: Displays a date range (or domain) label in the visualization, which is dynamically formatted based on the xDom signal and controlled by the showDomainSpanLabel flag.
  + Positioned on the right side of the chart with a dynamic appearance, allowing customization of visibility and positioning.

This section of the code is designed to make the chart interactive and visually dynamic, with user controls (buttons) and contextual information (date range labels) being rendered conditionally.

**1. Phase Backgrounds (phaseBackgrounds):**

{

"name": "phaseBackgrounds",

"description": "Background rect for phases",

"type": "rect",

"clip": true,

"zindex": 0,

"from": {"data": "phases"},

"encode": {

"update": {

"x": {"value": 0},

"x2": {"signal": "columnsWidth"},

"y": {

"signal": "scale('y', datum.id)"

},

"height": {

"signal": "bandwidth('y')"

},

"fill": {"value": "#dceaf7"},

"opacity": {"value": 0.3}

}

}

}

**Explanation:**

* **type: "rect"**: This mark draws a rectangle for the background of phases.
* **clip: true**: Ensures that the rectangle stays within the bounds of the visualization.
* **zindex: 0**: Places this background rectangle at the lowest z-index, ensuring other elements (like tasks) appear above it.
* **from: { "data": "phases" }**: This background rectangle is drawn for each item in the phases dataset.
* **encode**:
  + **x**: Sets the starting horizontal position at 0.
  + **x2**: Defines the rectangle’s right edge, extending to the full width of the columns (columnsWidth).
  + **y**: The vertical position is determined by the phase’s id, scaled using the y scale.
  + **height**: Uses the bandwidth('y') function to set the height of each phase background.
  + **fill**: The fill color is a light blue (#dceaf7).
  + **opacity**: The opacity is set to 0.3, making the background slightly transparent.

**2. Task Label Sizes (taskLabelSizes):**

{

"name": "taskLabelSizes",

"description": "Hidden label sizes to support tooltips when the task name doesn't completely fit",

"type": "text",

"clip": true,

"from": {"data": "yScale"},

"encode": {

"enter": {

"x": {"value": -100},

"y": {"value": -100},

"fill": {

"value": "transparent"

},

"text": {

"signal": "datum.task"

},

"fontSize": {"value": 11}

}

}

}

**Explanation:**

* **type: "text"**: This creates a text mark.
* **clip: true**: Clips any content that goes outside the bounds of the container.
* **from: { "data": "yScale" }**: Uses the data from the yScale to position text elements.
* **encode (enter)**:
  + **x, y**: Initially places the text off-screen (x: -100, y: -100) so it’s not visible in the chart.
  + **fill**: Sets the text color to transparent so it won't be rendered.
  + **text**: The text content is set to the task name (datum.task).
  + **fontSize**: The font size is set to 11 units for readability.

This mark is useful for measuring the task names in a hidden state to later calculate whether they fit within the allocated space or need a tooltip.

**3. Task Tooltips (taskTooltips):**

{

"type": "rect",

"name": "taskTooltips",

"description": "Hidden rect to support tooltips when the task name doesn't completely fit",

"from": {

"data": "taskLabelSizes"

},

"clip": true,

"zindex": 101,

"encode": {

"update": {

"x": {"value": -15},

"x2": {

"signal": "taskColumnWidth"

},

"y": {

"signal": "scale('y', datum.datum.id)"

},

"height": {

"signal": "bandwidth('y')"

},

"fill": {

"value": "transparent"

},

"tooltip": {

"signal": "datum.bounds.x2 - datum.bounds.x1 >= taskColumnWidth - 16 ? datum.datum.task : null"

},

"cursor": {

"signal": "datum.datum.phase == datum.datum.task ? 'pointer' : 'auto'"

}

}

}

}

**Explanation:**

* **type: "rect"**: A rectangle is used here for detecting the area where the task name might overflow.
* **from: { "data": "taskLabelSizes" }**: This uses data from the previously defined taskLabelSizes to define where the tooltip area will be.
* **clip: true**: Ensures that the rectangle is confined to the bounds of the container.
* **zindex: 101**: This makes sure the tooltip rectangle is layered above other elements.
* **encode (update)**:
  + **x**: Positions the rectangle slightly off to the left (-15).
  + **x2**: Sets the right edge based on taskColumnWidth, which represents the available width for the task name.
  + **y**: Positions the rectangle vertically according to the task's id, scaled by the y scale.
  + **height**: The height matches the task’s height, determined by bandwidth('y').
  + **fill**: The rectangle is transparent (fill: "transparent"), as it’s only used for interaction detection.
  + **tooltip**: If the task name overflows the available column width (calculated by datum.bounds.x2 - datum.bounds.x1 >= taskColumnWidth - 16), it triggers the display of the tooltip with the task name.
  + **cursor**: Changes the cursor to a pointer when hovering over the task area if it’s a task phase (datum.datum.phase == datum.datum.task).

This mark is essential for enabling tooltips to appear when task names do not fit within the allocated space in the chart.

**4. Column Holder (columnHolder):**

{

"type": "group",

"name": "columnHolder",

"style": "cell",

"layout": {

"padding": {

"signal": "columnPadding"

},

"bounds": "flush",

"align": "each"

},

"encode": {

"enter": {

"x": {"signal": "0"},

"stroke": {

"value": "transparent"

},

"width": {

"signal": "columnsWidth"

},

"height": {"signal": "height"}

}

}

}

**Explanation:**

* **type: "group"**: This is a grouping element that can contain other visual marks. It is used to organize related marks.
* **name: "columnHolder"**: This group is used for holding columns of tasks.
* **style: "cell"**: Assigns a cell-style layout to the group.
* **layout**:
  + **padding**: Adds padding using the columnPadding signal to space out the columns.
  + **bounds: "flush"**: Ensures the bounds of this group are tightly bound to its content.
  + **align: "each"**: Aligns the contents of each column.
* **encode (enter)**:
  + **x**: Sets the horizontal position of the column group to 0.
  + **stroke**: Sets the stroke (border) to transparent, meaning there is no visible border.
  + **width**: The width of the column is determined by the columnsWidth signal, which adapts to the chart size.
  + **height**: The height is set based on the height signal, ensuring the group fills the available space.

This mark serves as a container for individual columns in the visualization, likely containing multiple tasks or phases.

**Summary:**

* **Phase Backgrounds**: Light blue backgrounds for each phase.
* **Task Label Sizes**: A hidden text mark used to calculate and measure task name overflow for tooltips.
* **Task Tooltips**: Rectangular areas used to detect overflow and display task tooltips when necessary.
* **Column Holder**: A group element used to organize the columns and tasks, with configurable padding and dimensions.

These components work together to support the layout, interaction, and visual elements of a Gantt chart, making sure task names fit, overflow is detected, and tooltips appear when necessary.

**1. Group Mark - portfolioColumnWidth:**

{

"type": "group",

"name": "portfolioColumnWidth",

"style": "cell",

"title": {

"text": "Portfolio",

"anchor": "middle",

"frame": "group",

"align": "right"

},

"encode": {

"update": {

"width": {

"signal": "portfolioColumnWidth"

},

"stroke": {

"value": "transparent"

},

"height": {

"signal": "height"

}

}

},

"marks": [

{

"type": "text",

"style": "col",

"clip": true,

"from": {

"data": "yScale"

},

"encode": {

"update": {

"align": {

"value": "right"

},

"x": {

"signal": "portfolioColumnWidth"

},

"y": {

"signal": "scale('y', datum.id) + bandwidth('y') / 2"

},

"text": {

"signal": "datum.portfolio"

},

"fontWeight": {

"signal": "datum.phase == datum.task ? 'bold' : 'normal'"

},

"font": {

"signal": "datum.phase == datum.task ? 'Arial' : 'Segoe UI'"

},

"fill": {

"signal": "datum.phase == datum.task ? 'Black' : 'white'"

}

}

}

}

]

}

**Explanation:**

**Group Mark - portfolioColumnWidth:**

* **type: "group"**: The main container is a group mark that allows grouping other visual elements (in this case, the text label inside it).
* **name: "portfolioColumnWidth"**: This assigns a name to the group so it can be referenced in other parts of the specification.
* **style: "cell"**: Applies the "cell" style to the group, which might include layout properties like padding or alignment. The "cell" style could be predefined elsewhere in the specification.
* **title:**
  + **text: "Portfolio"`**: Sets the title to "Portfolio".
  + **anchor: "middle"`**: Aligns the title horizontally in the middle of the group.
  + **frame: "group"`**: The title is placed relative to the group itself.
  + **align: "right"`**: Aligns the title to the right side of the group.

**encode (update):**

* **width**: The group's width is determined by the signal portfolioColumnWidth. This signal likely represents the dynamically calculated width for the portfolio column.
* **stroke**: The group has no visible stroke (transparent).
* **height**: The group's height is determined by the height signal, which could represent the overall height of the visualization or container.

**Text Mark Inside the Group:**

* **type: "text"`**: The mark used is a text mark that displays text content.
* **style: "col"`**: This applies the "col" style to the text, likely defining properties like font size, color, etc.
* **clip: true**: Ensures that any text that overflows the bounds of the group is clipped (hidden).
* **from:**
  + \*\*data: "yScale": The text mark is based on data from yScale`, which is a transformation or data set containing the task data.

**encode (update) for the Text Mark:**

* **align:**
  + \*\*value: "right"`: Aligns the text to the right.
* **x:**
  + \*\*signal: "portfolioColumnWidth": The x-position of the text is set based on the portfolioColumnWidth` signal, which determines how far across the canvas the text is placed.
* **y:**
  + \*\*signal: "scale('y', datum.id) + bandwidth('y') / 2": The y-position is set based on a scale function (scale('y', datum.id)) that places the text based on the task's id, with an additional offset defined by bandwidth('y') / 2` to center the text vertically within the allocated space.
* **text:**
  + \*\*signal: "datum.portfolio": The text displayed is pulled from the portfolio` field of the data (likely representing the name or title of the portfolio or task).
* **fontWeight:**
  + **signal: "datum.phase == datum.task ? 'bold' : 'normal'"**: The font weight is set to "bold" if the phase matches the task, otherwise it is "normal".
* **font:**
  + **signal: "datum.phase == datum.task ? 'Arial' : 'Segoe UI'"**: The font family is set to "Arial" if the phase matches the task, otherwise it uses "Segoe UI".
* **fill:**
  + **signal: "datum.phase == datum.task ? 'Black' : 'white'"**: The text color is set to "black" if the phase matches the task, otherwise it is "white".

**Purpose:**

This group and text mark combination is used to display the **portfolio** label or task names inside the portfolio column:

* The **group** (portfolioColumnWidth) contains the portfolio title and ensures that it scales dynamically based on available space.
* The **text** inside the group will display the portfolio name (datum.portfolio), which is dynamically positioned and styled based on conditions like whether the phase matches the task.
* The **font weight**, **font**, and **text color** are conditionally adjusted based on the phase and task relationship, providing visual differentiation.

This structure is helpful for creating a dynamic, responsive portfolio or task list where each task is clearly labeled, and the appearance changes based on the task's status or relationship to its phase.

This part of the Vega specification defines three columns in a visual representation of tasks, each corresponding to different attributes of the tasks: **System**, **Start**, and **End**. These columns are defined as **group marks** with corresponding **text** and **symbol** marks. Here's a breakdown of the specification:

**1. Task Column (System) - taskColumnWidth:**

This column represents the system or task-related data.

**Group Mark: taskColumnWidth:**

* **Type**: "group" — This is a container for organizing marks (such as text and symbols).
* **Style**: "cell" — This could apply predefined styles for visual appearance.
* **Title**:
  + **Text**: "System" — The title of the column is "System".
  + **Anchor**: "start" — The title is aligned to the left.
  + **Frame**: "group" — The title is placed relative to the group's frame.
  + **Align**: "left" — Aligns the title to the left of the group.
  + **dx**: 16 — Shifts the title horizontally by 16 pixels.

**Encode (Group Mark):**

* **Width**: Dynamically calculated using taskColumnWidth (a signal representing the column's width).
* **Height**: Dynamically calculated using height.
* **Stroke**: No visible stroke (transparent).

**Marks Inside taskColumnWidth:**

1. **Text Mark** (for the task label):
   * **Text**: The task's name is displayed (datum.task), and it is conditionally transformed using upper(datum.task) if it matches the current phase.
   * **Font Weight**: The task name is bolded if the task matches the current phase (datum.phase == datum.task).
   * **Font**: The font used is "Arial" for the task in the current phase, and "Segoe UI" for others.
   * **Color**: The text color changes based on whether the task is hovered over (itemHovered.id).
2. **Symbol Mark** (for phase indicator):
   * **Shape**: The shape changes based on whether the task is collapsed or expanded. If the task is collapsed, a downward triangle is used; if expanded, a rightward triangle is used.
   * **Color**: The color is determined by whether the task is hovered over (itemHovered.id) and if it matches the current phase.
   * **Size**: The size of the symbol is set to 90, likely indicating the size of the phase indicator.

**2. Start Column - startColumnWidth:**

This column represents the **start date** of tasks.

**Group Mark: startColumnWidth:**

* **Type**: "group"
* **Style**: "cell"
* **Title**: "Start"
  + Aligned to the left, with no additional horizontal shift (dx is not specified).

**Encode (Group Mark):**

* **Width**: Dynamically calculated using startColumnWidth.
* **Height**: Dynamically calculated using height.
* **Stroke**: No visible stroke (transparent).

**Marks Inside startColumnWidth:**

1. **Text Mark**:
   * **Text**: The start date of the task is formatted using timeFormat(datum.start, '%m/%d/%y').
   * **Font**: Uses "Arial" for tasks in the current phase, "Segoe UI" for others.
   * **Font Weight**: The text is bold if the task matches the current phase.
   * **Color**: The text color is determined by whether the task is hovered over (itemHovered.id).

**3. End Column - endColumnWidth:**

This column represents the **end date** of tasks.

**Group Mark: endColumnWidth:**

* **Type**: "group"
* **Style**: "cell"
* **Title**: "End"
  + Aligned to the left, with no additional horizontal shift (dx is not specified).

**Encode (Group Mark):**

* **Width**: Dynamically calculated using endColumnWidth.
* **Height**: Dynamically calculated using height.
* **Stroke**: No visible stroke (transparent).

**Marks Inside endColumnWidth:**

1. **Text Mark**:
   * **Text**: The end date of the task is formatted using timeFormat(datum.labelEnd, '%m/%d/%y').
   * **Font**: Uses "Arial" for tasks in the current phase, "Segoe UI" for others.
   * **Font Weight**: The text is bold if the task matches the current phase.
   * **Color**: The text color is determined by whether the task is hovered over (itemHovered.id).

**Summary:**

* **System Column** (taskColumnWidth): Displays the task's name, with dynamic styling for the font weight, font type, and color based on the phase and hover state. It also contains a symbol that changes shape based on whether the task is collapsed or expanded.
* **Start Column** (startColumnWidth): Displays the start date of the task, formatted as MM/DD/YYYY.
* **End Column** (endColumnWidth): Displays the end date of the task, formatted similarly.

All columns share dynamic properties like width and height that adjust according to signals (taskColumnWidth, startColumnWidth, endColumnWidth, and height). The visual elements also respond to interactions, such as changing colors when tasks are hovered over, making the visualization more interactive.

This part of the Vega specification defines a **ganttContainer** group, which holds various marks for displaying task-related information in a Gantt chart-like format. Let's break it down:

**Gantt Container - ganttContainer:**

This **group mark** defines the container where all other visual elements (like tasks, labels, and completion indicators) are rendered.

**Encode (Gantt Container):**

* **x**: The horizontal position of the container is determined by the signal columnsWidth.
* **y**: The vertical position is set to 0.
* **Clip**: The clip signal ensures that any overflow content within the container is hidden.
* **Height**: The height is dynamically set by the signal height.
* **Width**: The width is dynamically set by the signal ganttWidth.
* **Fill**: The background color of the container is set to transparent.

**Marks Inside ganttContainer:**

Inside the container, two specific marks are defined:

**1. Completion Label Sizes - completionLabelSizes:**

This **text mark** is used for rendering the completion label of each task (though it is set to be transparent initially).

* **From**: This mark uses data from tasks.
* **Text**: The content of the label is pulled from datum.completionLabel.
* **Fill**: Set to transparent, so the label isn't visible in the mark itself.

**2. Task Labels - taskLabels:**

This **text mark** is used to render the task names along with their durations (in days).

**Encode (Task Labels):**

* **x**: The horizontal position of the label is determined by the x scale applied to the end field, which represents the end date of the task.
* **Align**: The text is aligned to the left ('left').
* **dx**: A horizontal offset is applied to the label (dayBandwidth/2 + 5), which is likely used to ensure that the label doesn't overlap with other elements or tasks.
* **y**: The vertical position of the label is determined by whether the task corresponds to the current phase (datum.phase == datum.task). If so, the vertical position is slightly adjusted (-2).
* **dy**: The vertical displacement is set to half of the band width (bandwidth('y')/2), centering the text vertically relative to the task.
* **Fill**: The fill color for the text is determined based on whether the task is hovered over (itemHovered.id). If hovered, the color is dynamically adjusted using a color scale (hsl(scale('cDark', datum.task))), with some lightness change. Otherwise, it uses a textColour.
* **Text**: The task label text is dynamically generated with the task name and the number of days (datum.task + ' ('+ datum.days+' d'+')').

**Summary:**

* The **ganttContainer** group holds the visual elements for a Gantt chart-like display.
* Inside the container:
  1. **completionLabelSizes** is used to represent task completion labels but is invisible (transparent).
  2. **taskLabels** renders task names and their durations, dynamically positioning the labels based on task data and adjusting their appearance (color, font weight) based on hover and phase states.

This setup allows for a clear representation of tasks with information like their name and duration, and incorporates interactive hover effects to highlight specific tasks.

This part of the Vega specification defines three key visual components: dependency lines, a rule to indicate "today's" date, and a text label for "Today". Let’s break down each component:

**1. Dependency Lines - dependencyLinesFacet**

This section visualizes relationships between tasks or items based on dependency data. It defines a group to hold lines representing dependencies between tasks.

**Facet Group - dependencyLinesFacet:**

* **Facet**: The dependencyLinesFacet groups the data in dependencyLines by the fields id and sourceId. This helps organize the data by task dependencies, ensuring that lines are drawn correctly between source and target tasks.

**Marks Inside dependencyLinesFacet:**

* **Line Mark**: Represents the dependency between two tasks with a line.
  + **x & y**: The position of the line is determined by the datum.value[0] and datum.value[1] values, which represent the starting and ending points of the line.
  + **Stroke**: The line color is set to #888888 by default, but it changes when a task is hovered. If a task is hovered, the color is dynamically updated using the scale('cDark', datum.task) function.
  + **Stroke Width**: The width of the line is set to 1 by default. If the line is hovered (i.e., datum.id == itemHovered.id), the width increases to 1.5 to make the line more prominent.
  + **Interpolation**: The line uses a linear interpolation, meaning it’s a straight line between the source and target.
  + **Stroke Join & Cap**: The line has beveled corners and rounded ends.
  + **Defined**: Set to true, ensuring that the line is visible as long as the data is valid.

**2. Today Rule - todayRule**

This rule represents a visual indication of the current date (today). It is drawn as a dashed vertical line.

**Properties:**

* **x**: The x position is calculated using scale('x', today + oneDay/2), which positions the line at the "today" date.
* **y2**: The vertical extent of the line is determined by a dynamic signal scaledHeight, which ensures the line covers the full height of the chart.
* **Stroke**: The line is blue (#377eb9), and it's drawn with dashed strokes ([2, 2]).
* **Stroke Width**: The line has a width of 1.
* **Opacity**: The opacity is set to 0.8 to make it visible but slightly transparent.

**3. Today Text - todayText**

This is a label that displays the word "Today" next to the "today" line.

**Properties:**

* **x**: The horizontal position of the label is aligned with the today date using the same scale('x', today + oneDay/2) calculation as the rule.
* **Text**: The label text is set to "Today".
* **Angle**: The text is rotated by 90 degrees (angle: 90), which likely makes it vertical.
* **Baseline**: The baseline of the text is aligned to the bottom.
* **dx & dy**: These values adjust the positioning of the text label relative to the "today" line.
  + dx: 10 shifts the label slightly to the right.
  + dy: -4 shifts the label upwards.
* **Fill**: The label text is blue (#377eb9), matching the color of the "today" rule line.
* **Opacity**: The opacity of the label is set to 0.7, making it slightly transparent.

**Summary:**

* **Dependency Lines**: Represents task dependencies with lines that adjust in appearance based on hover interactions.
* **Today Rule**: Draws a vertical dashed line to indicate the current date.
* **Today Text**: Displays the label "Today" near the today rule, indicating the current date.

This setup ensures that users can visually track dependencies between tasks while also clearly seeing the current date ("Today") highlighted on the chart.

This section describes a visualization for task bars in a Gantt chart, highlighting both the tasks and their completion percentages. The taskBars group contains several important details to render the task bars with proper styling, tooltips, and interactivity. Let's break it down:

**Task Bars Group (taskBars)**

* **Data Source**: The group is drawn from the "tasks" data.
* **Positioning & Dimensions**:
  + **x** and **x2**: These values represent the start and end of the task on the horizontal axis (x), based on the start and end values of each task.
  + **yc**: This determines the vertical center (yc) of the task bar, positioned using the y scale and adjusted by half of the task's bandwidth (height).
  + **Height**: The height of the task bar is dynamically calculated using bandwidth('y') minus padding (yPaddingInner).
* **Tooltip**:
  + **Tooltip Content**: A tooltip shows detailed information when hovering over a task. This includes the task phase, task name, description, start date, and end date. The showTooltips flag determines if tooltips should be displayed, and tooltips are shown only when down == null (presumably indicating the absence of a click or active selection).
* **Fill & Stroke**:
  + **Fill Color**: The task bar’s fill color is based on the task’s phase. When a task or its dependencies are hovered, the fill color lightens (merge(hsl(scale('cLight', datum.phase)), {l:0.65})), otherwise, it uses the standard phase-based color (scale('cLight', datum.phase)).
  + **Stroke Color**: Similarly, the stroke color adjusts based on whether the task is hovered or is part of the hovered task's dependencies. If hovered, it darkens the color (merge(hsl(scale('cDark', datum.task)), {l:0.40})); otherwise, it uses the standard task-based color (scale('cDark', datum.task)).
  + **Stroke Width**: If a task is hovered or part of the hovered task's dependencies, the stroke width is increased to 1.5 (otherwise, it’s 1).
* **Corner Radius**: The task bars have rounded corners with a radius of 5 for both top-left and bottom-left corners.
* **Z-Index**: Set to 101, ensuring that the task bars are rendered above other elements.

**Transform: Lookup for Completion Labels (completionLabelSizes)**

This transform performs a lookup to get the bounds of completion labels for each task. The fields datum.id, bounds.x1, and bounds.x2 are used to map the start and end positions for each task, allowing the completion bars to be positioned accurately.

* **Key**: It looks up the task's ID (datum.id).
* **Values**: It retrieves the completion label's x1 and x2 bounds (likely representing the left and right edges of the completion label, which helps position the completion percentage bar correctly).

**Marks: Completion Fill Bar**

A **rect** (rectangle) is drawn to represent the task's completion percentage within the task bar.

**Properties:**

* **x** and **y**: The completion bar starts at 0, meaning it fills from the left side of the task bar.
* **Height**: The completion bar’s height is determined by the height of the task bar.
* **Width**: The width of the completion bar is calculated based on the task’s completion percentage ((item.mark.group.width/100) \* item.mark.group.datum.completion), scaling the width proportionally to the completion percentage.
* **Fill**: The fill color is determined by the task's ID and phase, similar to the task bar's fill. The color adjusts when hovering over the task or its dependencies.
* **Stroke Width**: Set to 0 to avoid any borders around the completion bar.
* **Corner Radius**: The completion bar also has rounded corners at the bottom-left and top-left to match the task bar’s styling.

**Summary:**

* **Task Bars**: Represent the main task bars on the Gantt chart. They are positioned based on the task's start and end times, with color and stroke effects that change on hover.
* **Percent Complete Bar**: A completion bar inside each task bar shows how much of the task is complete, with the width based on the completion percentage.
* **Hover Effects**: Both the task bars and completion bars have hover effects that change their color and stroke width for better interactivity.
* **Tooltips**: When hovering over a task, tooltips appear with detailed task information like phase, task name, and dates.

This design ensures that users can easily see the task's status, including its completion progress, while also offering interactivity through hover effects and tooltips.

This section defines the visual elements for the **phase outlines** and **phase fills**, which are used to represent the phases of tasks in the Gantt chart. Here's a breakdown of what these components do:

**Phase Outline (phaseOutline)**

The phaseOutline represents the outer boundary of the phase bar for each task, making it visually distinct.

* **Type**: path — A path element is used to create the outline for the phase bar.
* **Data Source**: The path data comes from phasePaths, which likely defines the coordinates for the phase boundaries.

**Attributes:**

* **Path**:
  + The path is dynamically generated from the datum.phasePath, which probably holds the SVG path commands to draw the phase bar.
* **Fill**:
  + The fill color is determined by the task's current phase. When the phase outline or its dependencies are hovered over, the fill color lightens (merge(hsl(scale('cLight', datum.currenttask)), {l:0.65})). Otherwise, it uses a phase-based color (scale('cLight', datum.currenttask)).
* **Stroke**:
  + The stroke (border) color adjusts based on the hovered task. If hovered or if a dependency is hovered, the stroke darkens (merge(hsl(scale('cDark', datum.task)), {l:0.40})), otherwise, the stroke color is based on the task (scale('cDark', datum.task)).
* **Stroke Width**:
  + When the phase outline is hovered, the stroke width increases to 1.5, otherwise it is 1.
* **Tooltip**:
  + If showTooltips is true and the element is not clicked (i.e., down == null), a tooltip will appear showing phase information, including the phase name, description, start and end dates.
* **Cursor**:
  + The cursor changes to a pointer when hovering over the phase outline, indicating interactivity.

**Phase Group (phaseGroup)**

The phaseGroup is a grouping element that holds the x and y coordinates for each phase's clipping fills. It is used to structure and position the phase visualizations.

* **Clip**: true — The contents of this group will be clipped, ensuring that they don't extend beyond their defined boundaries.

**Attributes:**

* **Position**:
  + The x and x2 values define the horizontal positioning of the phase, using the x scale and task's start and end times.
* **YC**:
  + The vertical center (yc) of the phase group is calculated using the y scale and adjusting by half of the bandwidth('y'), positioning it within the vertical space allocated for each task.
* **Height**:
  + The height of the phase group is determined by the bandwidth('y') (the vertical space for each task) minus some inner padding (yPaddingInner).

**Phase Fills (phaseFills)**

The phaseFills are used to represent the completion percentage for each phase within the task.

**Attributes:**

* **Type**: rect — A rectangle is used to represent the phase completion.
* **Clipping Path**:
  + A dynamic clipping path is defined using a signal that describes the shape of the clipping area for the phase. The path is constructed with SVG commands (M 0 0 to L), which ensures that the fill is clipped appropriately according to the phase's start and end times. This path also adjusts the shape by adding a phaseSymbolHeight and phaseSymbolWidth, which likely control the appearance of the phase bar and how it is clipped based on the phase's completion.
* **Width**:
  + The width of the fill represents the completion percentage for the phase, calculated as a fraction of the total width of the phase ((item.mark.group.width / 100) \* item.mark.group.datum.completion).
* **Fill**:
  + The fill color of the phase is based on the task's phase color (scale('cDark', datum.task)). The fill lightens when hovered or when dependencies are hovered (merge(hsl(scale('cDark', item.mark.group.datum.task)), {l:0.40})).
* **Stroke**:
  + The stroke color for the phase fill is set to red, and its width is 0, meaning no visible border around the phase.

**Summary**

1. **Phase Outline**:
   * Draws an outline for each phase bar with interactivity (hover effects, tooltips) and color adjustments based on hover or dependencies.
2. **Phase Group**:
   * Organizes the phase paths and applies the necessary clipping and positioning, allowing for accurate phase bar display within the Gantt chart.
3. **Phase Fills**:
   * Represents the phase completion with a fill that visually shows how much of the phase is completed, with dynamic width based on completion percentage. This fill adjusts with interactive hover effects and tooltips.

The result is a set of elements that not only display the phases of tasks but also allow users to interact with them by hovering to get more information and visually showing completion progress. The dynamic color and stroke adjustments add a layer of interactivity and visual clarity, especially in distinguishing dependencies and hover states.

This section includes two elements: the **dependency arrows** and the **task selector**, each playing a role in enhancing the functionality and interactivity of the Gantt chart. Here’s a breakdown of what these components do:

**Dependency Arrows (taskDependencyArrowsymbol)**

The taskDependencyArrowsymbol represents arrows that indicate dependencies between tasks. These arrows are visual indicators showing the relationships between tasks in a Gantt chart.

**Attributes:**

* **Type**: symbol — This uses a symbol (likely a vector shape) to represent the dependency arrows.
* **Data Source**:
  + The arrows are generated from dependencyArrows, which likely contains the data about the dependencies between tasks.
* **Shape**:
  + The arrow shape used is defined as "triangle-right", creating a right-facing triangle that symbolizes the dependency direction.
* **Position**:
  + **X**: The horizontal position of the arrow is determined by the task's start time, using the x scale. It is offset by a small value calculated from the square root of arrowSymbolSize, adjusted with dayBandwidth/2 + 1.
  + **Y**: The vertical position is calculated using the y scale for each task, positioning the arrow at the center of the task's row (bandwidth('y')/2).
* **Fill**:
  + The fill color of the arrow is conditionally changed when the task is hovered over. If the datum.id of the arrow matches the itemHovered.id, the color darkens (based on the task's phase color with a slight adjustment in lightness). Otherwise, the default color is #6a6a6a.
* **Size**:
  + The size of the arrow is dynamically set with arrowSymbolSize, which adjusts based on how large the symbol should appear.

**Task Selector (taskSelector)**

The taskSelector is a hidden interactive element that allows for phase expansion and collapse within the Gantt chart.

**Attributes:**

* **Type**: rect — A rectangle is used as the base shape for the task selector.
* **Clip**:
  + The task selector is clipped, ensuring that it does not overflow and is contained within the visible bounds of the Gantt chart.
* **Z-index**:
  + The zindex of 99 ensures that the task selector is rendered above other elements, allowing it to be interacted with more easily.
* **Position**:
  + **X**: The rectangle starts from -15 on the x-axis, and its width (x2) is determined by the columnsWidth signal, which likely calculates the overall width of the columns in the chart.
  + **Y**: The vertical position is determined by the y scale, which places the rectangle at the correct vertical level corresponding to the task (datum.id).
  + **Height**: The height is set by bandwidth('y'), which corresponds to the task's allocated height in the Gantt chart.
* **Fill**:
  + The fill is set to transparent, so the rectangle is not visible, but it still functions as an interactive area for detecting clicks.
* **Cursor**:
  + The cursor property changes to pointer if the task is a phase (when datum.phase == datum.task), indicating that the task is expandable or collapsible. Otherwise, the cursor remains as auto.

**Summary**

1. **Dependency Arrows**:
   * These arrows visually represent dependencies between tasks, with their positions and appearance dynamically adjusted based on the task data and interactions (hovering). The arrows show the relationship between tasks and help users visually understand task flow.
2. **Task Selector**:
   * This hidden interactive rectangle supports expanding and collapsing phases in the Gantt chart. It's an invisible but interactive area that allows users to click on phase bars for expansion or collapse, with the cursor changing to a pointer to signal interactivity.

The taskDependencyArrowsymbol and taskSelector elements combine interactivity and clarity, enhancing the Gantt chart's usability and making it easier to interact with and visualize dependencies and task phases. The dynamic adjustments in size, color, and cursor states help users quickly identify relationships between tasks and interact with the chart.

This JSON configuration defines a Gantt chart's structure, including axes, scales, and visual style elements. Here’s a breakdown of the various components:

**1. Axis Clipper (axisClipper)**

* **Purpose**: This group element clips the axis rendering to the defined size.
* **Attributes**:
  + **Clip**: Ensures that elements that overflow this group are hidden.
  + **Width and Height**: Set dynamically using columnsWidth and height signals, respectively.
  + **Stroke**: Set to transparent to avoid visible borders.

**Axes Configuration:**

* **Y-Axis**:
  + **Orientation**: Positioned to the right ("right").
  + **Ticks**: The axis ticks are positioned to the left and are adjusted with the x2 signal.
  + **Tick Color**: The color of the ticks is set to #f1f1f1 (a light gray).
  + **Label Overlap**: Disabled, as the axis doesn’t display labels.
* **X-Axis (Date Axis)**:
  + **Description**: This axis displays the timeline of the Gantt chart (dates).
  + **Scale**: Uses a time scale ("x").
  + **Orientation**: Positioned at the top ("top").
  + **Tick Size**: Set to 15, determining the size of each tick.
  + **Label Padding**: Negative padding (-12) is applied to adjust label positioning.
  + **Tick Format**: The tick values will represent dates, formatted according to the time format type.
  + **Tick Count**: Adjusted dynamically depending on the dayBandwidthRound value — it will show days if dayBandwidthRound is greater than or equal to the minimum year bandwidth, or months otherwise.
  + **Label Logic**: Custom logic to display labels like "FY" (Fiscal Year) for specific dates.

**2. Scales**

Scales define how data values are mapped to visual elements on the chart:

* **X Scale (x)**:
  + **Type**: Time scale for mapping date/time data to the horizontal axis.
  + **Domain**: Bound by the xDom signal.
  + **Range**: Mapped to the width of the chart (ganttWidth).
* **Y Scale (y)**:
  + **Type**: Band scale for positioning tasks vertically.
  + **Domain**: Based on the id field in the yScale dataset, with tasks ordered by finalSort.
  + **Range**: Adjusted dynamically by the yRange signal.
* **Color Scales (cDark and cLight)**:
  + **Type**: Ordinal scales, which are used to assign colors to tasks and phases.
  + **Range**: Mapped to color arrays coloursDark and coloursLight.
  + **Domain**: Tasks are mapped to specific colors based on their respective task IDs.

**3. Configuration**

This section sets up general appearance and style for the chart:

* **Font Settings**:
  + **Font**: Segoe UI is set for both the general text and axis labels.
  + **Font Size**: General font size for text and axis labels is set to 10px, while the font size for column labels is slightly larger at 11px.
* **View Configuration**:
  + **Stroke**: The chart's main view has a transparent stroke (no border).
* **Text Style**:
  + **Font and Size**: The text uses Segoe UI at 10px for readability.
  + **Baseline**: The text baseline is set to "middle," ensuring proper vertical alignment.
* **Axis Titles**:
  + **Title Color**: The color of axis titles is controlled by the textColour signal.

**Summary**

This configuration effectively sets up the visual elements of a Gantt chart. It creates a time-based X-axis and a band-based Y-axis for displaying tasks, with the ability to dynamically adjust based on the bandwidth for each task. It also incorporates conditional rendering logic for ticks and labels on the X-axis based on time (day or month). Colors are applied to tasks and phases using ordinal scales, and it uses a transparent clip to limit overflow in certain areas of the chart. Finally, the overall style uses a clean, modern font (Segoe UI) with subtle color adjustments for text and labels.

Top of Form

Bottom of Form