ChatGPT

Reusable Interactive Chart Viewer Implementation Guide

Supported Chart Types and Views

To cover a broad range of personal finance use cases, the chart viewer will support multiple chart types. These include time-series charts (e.g. **line** and **area** plots for trends over time), categorical charts (e.g. **bar** charts for comparisons, **pie** and **treemap** charts for part-to-whole visuals), and correlation charts (e.g. **scatter** or **bubble** plots for relationships). Each chart type addresses different analysis needs – for example, line/area charts show spending over time, bar charts compare categories, pie/treemaps show breakdown of a budget, and scatter/bubble charts might plot expenses vs. income or other metric correlations.

We will also incorporate **financial-specific views** that combine or customize these base chart types. Examples include a **cumulative spend vs. budget** line/area chart (plotting cumulative expenses against a target budget line), or a **revenue vs. expenses** comparison (dual-line chart or side-by-side bar chart). The chart library or implementation chosen must be flexible enough to handle at least ~10+ distinct chart types (as Recharts, for example, does [1]) to ensure all these scenarios are covered. Each chart type will be implemented as a reusable React component or configuration, so the Chart Viewer can render the appropriate visualization based on a spec.

Interactive behavior (tooltip on hover, legend toggling, etc.) will be supported for all chart types. For example, time series charts will show data point details on hover, categorical charts will allow toggling series via the legend, and pie charts will display slice details. This interactivity will be handled via the charting library's built-in features or custom event handlers. In summary, the component will serve as a unified chart container that can render **line, area, bar, column, pie, donut, treemap, scatter, bubble, radar** (if needed) and other charts, with easy expansion for future types [2].

Chart Specification Schema and Template System

At the core of the Chart Viewer is a **JSON-based chart specification** (or "chart spec") that defines what data to show and how to visualize it. This spec-driven approach makes the chart component highly reusable and configurable. We will design a **JSON schema** for chart specs with the following key fields:

- `queryRef` – Reference to the data source (e.g. an ID of a saved query or dataset to pull data from).
- `type` – The chart type (e.g. `"line"`, `"bar"`, `"pie"`, `"area"`, `"scatter"`, `"treemap"`, etc.).
- `axes` – Configuration for axes (if applicable), such as the x-axis and y-axis mappings. This includes which data field maps to each axis, axis labels, and scaling or formatting (e.g. time vs linear scale).
- `encoding` – Visual encoding definitions mapping data fields to visual elements like color, size, or shape. For example, encoding might specify which field determines slice categories in a pie chart or the grouping field for bar colors.

- `colors` – Optional color settings, such as a palette name or explicit colors for series/categories (useful to override defaults).
- `annotations` – Optional annotations to overlay on the chart, such as target lines, trend lines, or highlight regions (each annotation can specify type, position/value, and label).

Below is a TypeScript interface summarizing this schema (for clarity), which will guide both the JSON structure and our implementation:

```
interface ChartSpec {
  queryRef: string;
  type: 'line' | 'area' | 'bar' | 'pie' | 'treemap' | 'scatter' | 'bubble' |
string;
  axes?: {
    x?: { field: string; label?: string; scale?: 'time' | 'linear' | 'log';
format?: string };
    y?: { field: string; label?: string; scale?: 'linear' | 'log'; format?:
string };
    y2?: { field: string; label?: string; scale?: 'linear' | 'log'; format?:
string };  // for dual-axis charts if needed
  };
  encoding?: {
    color?: { field: string; palette?: string };    // e.g. field for series or
categories
    size?: { field: string };                       // e.g. field for bubble
size
    shape?: { field: string };                      // e.g. field for shape (if
used)
    value?: { field: string };                      // for pie/treemap: field
for numeric value
  };
  colors?: string[] | { [key: string]: string };    // optional explicit colors
or palette
  annotations?: Array<{
    type: 'line' | 'range' | 'point';
    axis?: 'x' | 'y';
    value: number | string | Date;
    label?: string;
  }>;
  title?: string;      // optional title for the chart
  description?: string; // optional description for accessibility or tooltip
}
```

JSON Schema – In JSON form, the schema would define these properties with types and required fields (for brevity, we use the TypeScript notation above to convey the shape). At minimum, `queryRef` and `type` are required. The `axes` field is usually only needed for cartesian charts (line, bar, scatter, etc.), whereas pie or treemap charts might ignore `axes` and instead rely on `encoding.category` / `encoding.value`.

The schema is flexible: for example, a pie chart spec would use `encoding.color` for the category field and `encoding.value` for the slice value field, and would not need `axes.x` / `y`. We will provide sensible defaults in the rendering logic for charts that don't use certain parts of the spec (e.g. ignore `axes` in a pie chart).

Using a JSON-based spec makes it easy to **persist and reuse chart configurations**. We will implement a robust **template system** around this schema:

- Users can create and save chart specs as **templates**. A template is essentially a saved `ChartSpec` JSON that can be reapplied to data.
- Templates can be **persisted per user** (each user can have their own collection of chart templates) and can also be marked as **global/shared**. For example, an admin or the system can provide global templates (available to all users) for common visualizations. In our data model (detailed later), we have a `chart_templates` collection/table with an `ownerUserId` and a `visibility` flag. If `visibility` is "public", the template is shared globally (or within the tenant); if "private", it's only visible to the owner. This approach is similar to how some BI tools allow admins to publish custom charts for all users [3].
- We will implement **default templates** that auto-apply by context. For instance, the app might define default chart specs for certain sections or data categories. If a user has not customized a chart for a given context, the default spec is used. These defaults could be keyed by a category/section identifier. For example, the "Budget" section might default to a cumulative budget vs actual area chart, while the "Expenses by Category" view defaults to a pie chart. The system can detect the context or data type of a query and choose an appropriate default template. Templates can even be selected based on query metadata – e.g., if the query returns time-series data, default to a line chart, but if it returns categorical breakdown, default to a bar or pie chart.

The JSON schema allows us to capture all of this in a single spec object. Here are a few **example chart spec JSONs** for illustration:

Example 1 – Line Chart (Time Series): Suppose we have a saved query for monthly spending totals. A line chart spec might look like:

```
{
  "queryRef": "monthly_spending",
  "type": "line",
  "axes": {
    "x": { "field": "month", "label": "Month", "scale": "time" },
    "y": { "field": "total_spend", "label": "Total Spend", "format":
"currency" }
  },
  "encoding": {
    "color": { "field": "category" }
  },
  "annotations": [
    { "type": "line", "axis": "y", "value": 1000, "label": "Monthly Budget" }
```

```
  ]
}
```

Explanation: This spec plots `total_spend` on the y-axis vs `month` on the x-axis (with a time scale). It encodes `category` by color, meaning if the query provides breakdown by category, each category gets a separate line (multiple series). An annotation draws a horizontal line at `y=1000` with label "Monthly Budget" to indicate a target budget level.

Example 2 – Bar Chart (Categorical): For a query that sums expenses by category in a given period:

```
{
  "queryRef": "spend_by_category_Q1",
  "type": "bar",
  "axes": {
    "x": { "field": "category", "label": "Category" },
    "y": { "field": "total_spend", "label": "Total Spend", "format":
"currency" }
  },
  "encoding": {
    "color": { "field": "category_group" }
  },
  "colors": ["#4E79A7", "#59A14F", "#E15759"],
  "annotations": []
}
```

Explanation: This spec produces a bar chart of `total_spend` per `category`. The x-axis is categorical (each category name), y-axis is the spend amount. We optionally use `encoding.color` with `category_group` (imagine categories might be grouped by necessity vs discretionary, for example) to color bars by a higher-level grouping. We also provide a custom color palette array for the groups. No specific annotations are added in this example. (If multiple time periods were compared, we could use grouped or stacked bars by extending the encoding.)

Example 3 – Pie Chart (Part-to-Whole): For a query that provides a breakdown of a monthly budget by category:

```
{
  "queryRef": "january_budget_breakdown",
  "type": "pie",
  "encoding": {
    "color": { "field": "category" },
    "value": { "field": "amount" }
  },
  "colors": {
    "Rent": "#4E79A7",
    "Food": "#59A14F",
```

```
    "Entertainment": "#E15759"
  },
  "annotations": []
}
```

Explanation: This spec defines a pie chart where each slice represents a `category` and its size is determined by the `amount` field (e.g. amount spent or budgeted for that category). We map `category` to slice colors (using a custom color mapping here for specific categories) and `amount` to the value of each slice. Axes are not applicable for a pie, so the `axes` key is omitted. The chart viewer will interpret this spec accordingly: iterate through data, sum or use each item's `amount`, and render slices labeled by `category` in the specified colors.

These examples demonstrate how the JSON spec can flexibly describe different chart configurations. The system will include **defaults** for omitted fields. For instance, if `colors` is not specified, a default palette will be applied based on the theme (with color-blind-friendly defaults, discussed later). If `encoding.color` is given but no `colors` list, the app can choose a palette automatically. If certain axis labels or formats are missing, we can infer them from field names or data types (e.g. add a currency symbol if the field name suggests money).

The template system will allow users to **persist** these chart specs. Each saved template or chart configuration will be stored in the database. Thanks to the JSON structure, templates are self-contained – they can be saved and later loaded to reconstruct the chart. This is analogous to how Vega-Lite uses JSON specs for charts [3], enabling users to define charts declaratively and share them. In our app, a user might save a custom spec as a template called "My Custom Budget Pie". That JSON spec (minus the data) can be reused with any compatible data source (i.e. any query that provides `category` and `amount` fields in this case).

Template defaults by category/section: The app can maintain a library of default templates keyed by a context key (e.g. `"monthly_budget"` could map to a line chart template, whereas `"category_breakdown"` maps to a pie chart template). When rendering, if a user hasn't customized that context, the viewer loads the default spec. Users can then override it by saving their own template and marking it as default (see below).

The template system will support the following features:

- **Save Template with Live Preview:** Users can adjust chart settings (type, fields, colors, etc.) in an editor UI and see the chart update live. Once satisfied, they can save the template. The JSON spec is stored in the backend (in `chart_templates` if meant for reuse, or in `charts` tied to a specific query if it's a one-off chart).
- **Template Gallery:** A gallery view will list saved templates. This includes personal templates and any global templates (with appropriate labels or filters). Each template entry can show a small preview (rendered from the spec on a sample dataset or using a cached image) so users recognize it. They can select a template from the gallery to apply it to a dataset.
- **Apply Template (Default vs Custom):** Users can toggle between default view and any custom templates for the data they're viewing. For example, on a "Monthly Summary" page, the default might be an area chart, but the user can switch to one of their saved templates (say a bar chart

version) via a dropdown or toggle. One template can be marked as **"default"** for that user and context – meaning it will load by default instead of the system default. In our data model, a `charts` record with `isDefault: true` serves this purpose. Only one chart per user/query would be marked default. If the user toggles back to the system default, we could either simply not use their saved spec or provide a way to "reset" default (which might just mean un-setting their `isDefault` chart).

- **Global/Shared Templates:** Templates with visibility "public" can be provided as starting points. For instance, the app might include built-in templates like "Basic Time Series Line Chart" or "Income vs Expense Comparison". All users can see and use these. They might be stored with a special `ownerUserId` (perhaps a system user or null) and `visibility = "public"`. Users can copy a global template into their own collection to customize if needed. This encourages reuse and a consistent style across the app [3].

Under the hood, the JSON schema will likely evolve. We might version the schema if needed, or validate it when loading (we can use a JSON Schema validator or TypeScript type checks). The chart rendering logic will parse the spec and instantiate the appropriate chart with the specified options.

Integration with Saved Queries and Real-Time Data Binding

The Chart Viewer's data comes from user-defined **saved queries** (or lists). A saved query represents a dataset – for example, "Monthly Spending by Category" or "Weekly Income vs Expenses". In the current Firebase/Firestore backend, a saved query could be represented as a Firestore document that includes the query parameters or references (e.g. which collection to query, filters, and any aggregation logic). In a future Postgres setup, it could correspond to a SQL query or a view. Regardless of implementation, each saved query has an identifier that we use as `queryRef` in the chart spec to fetch data.

Data Binding: When the Chart Viewer component loads, it will use the `queryRef` from the spec to retrieve the data, then render the chart. This will be designed to support **real-time updates** and reactivity:

- In a Firestore context, we can leverage Firestore's real-time capabilities. For example, if `queryRef` corresponds to a Firestore query (like "all transactions where month = January"), we can attach a snapshot listener. The chart will then update live whenever the underlying data changes (e.g. a new transaction is added) without a full page refresh. This fulfills the "real-time binding" requirement – the chart stays in sync with data changes.
- The component will likely use a React hook (e.g. a custom `useChartData(queryRef)` hook) to fetch and subscribe to data. On component mount, we initiate the data fetch:
- If using Firestore: call `onSnapshot(query, callback)` to get initial data and listen for updates.
- If using a REST/GraphQL API (as might be the case with Postgres), fetch the data via `fetch()` or an API client. Real-time in that scenario might involve WebSockets or polling if needed (for example, using something like Postgres LISTEN/NOTIFY via a server).
- **Re-execution on render:** Each time the chart spec or its `queryRef` changes (or the component mounts), the query will execute to get fresh data. We will also provide ways to manually refresh the chart (a refresh button to re-run the query on demand).

Caching Strategies for Expensive Queries: Some saved queries could be expensive (e.g. aggregating a year's worth of transactions). To keep the UI responsive and reduce load, we implement caching at multiple levels:

- **Client-side caching:** Using a library like React Query or SWR, we can cache the results of queries in memory. For example, React Query can cache data by `queryKey` (the `queryRef` can serve as a key) and provide stale-while-revalidate functionality. This means if the user navigates away and back to a chart, it can show cached data instantly and then refresh it in the background.
- **Firestore local cache:** Firestore SDK by default caches data it has fetched. If offline persistence is enabled, previously fetched query results are stored locally. Subsequent access to the same query can be served from the local cache first [4], then updated from the server. We will take advantage of this so that, for example, if a user frequently views the "monthly summary" chart, the data doesn't have to be fully refetched each time.
- **Result Storage (Write-time aggregation):** For very expensive queries, we consider maintaining pre-aggregated results. One approach is **write-time aggregation**, where each time underlying data changes, we update a stored summary. For instance, we could have a Cloud Function trigger on new transaction documents that updates a "monthly_totals" document. The chart query can then simply read that single document (fast) instead of scanning all transactions each time. This trades some write complexity for much faster reads [5]. Firebase now also supports some **read-time aggregation** like `sum()` and `count()` in queries [6], but those do not provide real-time updates or caching by themselves. Therefore, for charts that need live updates and involve lots of data, a write-time aggregation/caching strategy is ideal [5]. We might implement this for common metrics (e.g. total spend per month) – essentially storing a mini data mart in Firestore for the charts.
- **Server-side caching:** In a Postgres scenario, we might introduce a caching layer (e.g. Redis) or use materialized views. For example, a heavy query "spend vs budget for the year" could be backed by a materialized view that is refreshed daily. The Chart Viewer's API would then simply do `SELECT * FROM year_spend_vs_budget_view`. Alternatively, an application-level cache could store the JSON result of a query for a short time (say 10 minutes) – subsequent requests within that window get the cached result. This ensures snappy responses for dashboards.
- **Pagination or Windowing:** If a query can return a very large dataset (not typical for charts, but possible for very detailed plots), the chart component could fetch a window of data (e.g. last 12 months) and allow the user to request more via zooming or controls. This way, initial load is light. As the user drills in, more data can load on demand.

The combination of these strategies ensures that even if a query is complex, the app remains performant. For example, if a user opens a yearly trend chart, we might load cached results from last view immediately, then update if new data arrived. If no cache exists and data set is huge, we might show a loading indicator and possibly a message like "computing chart, please wait" while a cloud function aggregates the data in the background. Once computed, that result can be saved (cached) for subsequent loads.

Data Query Execution Example: In code, using Firestore as a baseline, it might look like:

```
// Pseudo-code for data fetching within the chart component
const [data, setData] = useState<DataPoint[]>([]);
const [loading, setLoading] = useState(true);

useEffect(() => {
```

```
  setLoading(true);
  const unsub = executeQuery(queryRef, (result) => {
    setData(result);
    setLoading(false);
  });
  return () => unsub(); // detach listener on unmount
}, [queryRef]);
```

Here, `executeQuery` would encapsulate either a Firestore `onSnapshot` call or an API fetch. If using Firestore, it could look like:

```
function executeQuery(queryRef, onResult) {
  const query = buildFirestoreQueryFromRef(queryRef);
  return onSnapshot(query, snapshot => {
    const result = snapshot.docs.map(doc => doc.data());
    onResult(result);
  });
}
```

For Postgres/REST, `executeQuery` could be an async function that fetches from an endpoint (and we wouldn't get continuous updates unless we set up web sockets or polling).

Binding to Chart: Once the data is fetched, the chart component passes it to the underlying chart renderer (e.g. a Chart.js or Recharts component). The viewer will need to transform the raw data into the format expected by the chart library – for instance, if using Recharts, an array of objects where each object has keys corresponding to axes or categories [7]. Our spec can help with this transformation, since it tells us which field is x, which is y, etc. We might write a small utility that given `ChartSpec` and raw data, returns a formatted dataset ready for the chart library.

Example: If data is an array of transactions and the spec says `axes.x.field = "month"` and `axes.y.field = "total_spend"`, we might group the data by month and sum spend (unless the query already did that). However, since saved queries likely already produce aggregated data (to avoid heavy front-end calculations), in most cases the data is already in the shape the chart needs (one record per x-axis value with the y value present, plus category breakdown if any). So transformation might be as simple as renaming fields or filtering out extra columns. In the Firestore context, saved queries could be realized as stored **Firestore queries or views** that handle the heavy lifting.

In summary, the Chart Viewer will **bind to saved query data** by executing the query reference, utilize **real-time updates** (especially with Firestore listeners) to keep charts live, and implement caching (client and server-side) to minimize redundant computations. This ensures that the charts are always up-to-date but also fast and efficient in a multi-user environment.

User Experience and Template Management Workflow

A major goal is to make the chart viewer **interactive and user-friendly**. This involves allowing users to customize charts, save their customizations as templates, and easily switch between different views. Below, we outline the UX flows and features in the app:

Creating & Editing Chart Templates (Live Preview)

Users will be able to create new chart templates through a visual editor interface. The typical flow:

1. **Open Chart Editor:** The user navigates to a section (say "My Charts" or via a specific dataset view) and clicks "Create Chart" or "Customize Chart". If they are on a page that already shows a default chart for a dataset, a "Customize" or "Edit Chart" button will open the editor for that chart.
2. **Select Data (Query):** If the user started from a dataset view, the query is already chosen (e.g. editing the chart for "Monthly Spending"). Otherwise, if starting blank, the first step is to choose a data source (from their saved queries or perhaps ad-hoc query builder).
3. **Configure Chart:** The user is presented with options to configure the chart. Key options include:
4. **Chart Type:** A dropdown or set of buttons to choose the type (line, bar, pie, etc.).
5. **Axes & Fields:** Depending on type, controls to select which fields map to X axis, Y axis, or category. For example, for a bar chart, user picks a field for X (category) and Y (value). This can be a select input populated with the fields available in the chosen query's results.
6. **Encodings:** Options for color grouping, size, etc. If the user wants to break a line chart by a second field (e.g. by account or category), they choose a field for "Color by". If making a scatter plot, they might choose fields for X, Y, and maybe one for point size.
7. **Appearance settings:** Color palette selection (with presets like a default theme, colorblind-friendly palette, etc.), toggles for things like "Stacked Bars" or "Show Legend", line style (solid/dashed), etc. These correspond to properties in the spec like `colors` array or certain encoding flags.
8. **Annotations:** UI to add an annotation (e.g. "Add reference line at value = ___ with label ___"). This updates the spec's `annotations` array.
9. **Title/Description:** Optionally, allow the user to name the chart (this could serve as the template name) and add a description. This can also double as chart title text if we display it on the chart.

As the user adjusts these controls, the chart preview updates in real-time. We will implement this by binding form state to the chart spec object. For example, if the user selects a different chart type, the `spec.type` state updates and the ChartViewer re-renders with the new type. If they change the color palette, we update `spec.colors` and re-render. This live preview is critical for a good UX, so the editor likely exists as a side-by-side or overlay view: controls on one side, chart on the other.

Under the hood, we can manage the editing state as a separate spec object (so as not to override the original until saved). We might utilize a state management library or context to hold the draft spec while editing.

1. **Save Template:** Once satisfied, the user clicks "Save". They will be prompted to name the template (if it's new) and choose save options:
2. Save just for this dataset (which would create a record in `charts` tied to the query).
3. Save as a reusable template (record in `chart_templates`), possibly with visibility choice (private or share globally).

4. Set as default for this dataset (a checkbox, which if checked will mark this chart as `isDefault` and override the app's default for this query in the future).

If the template is something general (not query-specific), we save it in `chart_templates`. If it's specifically a one-off customization for that query, we save in `charts` (with `queryId` set). In either case, the JSON spec is persisted in the backend. The preview chart the user was seeing is essentially the same spec, so WYSIWYG from preview to saved template.

1. **Feedback:** After saving, the user might see a confirmation and the new template appears in their gallery. If it was marked as default for that query, the next time they (or even immediately in the UI) view that query's chart, it will load this custom template by default.

During this process, we will validate inputs (e.g. ensure required fields for a given chart type are selected). If the user chooses a chart type and doesn't fill in a needed field, the UI can highlight it. Some complex scenarios (like a user choosing a pie chart for a query that has no obvious category/value fields) may need us to disable certain types or show a warning.

Template Gallery and Applying Templates

Users can view all their saved templates in a **Template Gallery** section. This will list templates perhaps as cards or a list with the template name and a thumbnail. For personal templates, they can edit or delete them. Global templates can be viewed (and possibly copied but not directly edited by normal users).

When viewing a particular dataset's chart, the user can **apply a different template** via a template selector. For example, on the "Monthly Spending" page, there could be a dropdown of available templates that are compatible with that data. This list might include: - The app's default template (always available). - Any user-saved charts for that dataset (from the `charts` collection where `queryId` matches). - Any global templates or the user's own templates that are marked as generally applicable (this is trickier – possibly we consider templates that have the same structure of fields required. This could be determined by checking if the fields in the template spec exist in the query's result schema. If they do, we can allow applying that template directly; if not, the template might be incompatible without modification).

If the user selects a template from the list, the Chart Viewer will load that template's spec (substituting the current `queryRef` if the template was generic) and re-render the chart. We'll provide a smooth transition, perhaps an animation or at least instantaneous update, so the user can compare views.

For example, suppose the default is a line chart but the user wants to see a bar chart comparison. If they have a saved bar chart template for this data (or a global one exists), they can choose it and the chart will redraw as a bar chart. They haven't overwritten the default; they are just viewing an alternative. They could then decide to save that as their default if they prefer it.

Default vs Custom Toggle: For each dataset view, we can implement a simple toggle or switch: "Use My Default" vs "System Default". If a user has a default template saved (`charts.isDefault` for that query and user), then "Use My Default" would be on by default. If they turn it off, we could temporarily show the system's default chart (without deleting their template). This gives a quick way to revert to the original view. It's essentially the same as selecting the app's default template from the gallery. We can implement it either as a toggle or just as an entry in the template dropdown (e.g. an item called "(Reset to default view)").

Additionally, if the user has multiple saved charts for the query (not all might be default), the gallery/dropdown approach covers switching between them. The “default” flag just determines which one loads initially.

Interactive Chart Controls: Filtering and Drilldowns

The Chart Viewer will support interactive filtering and drill-downs to make the charts exploratory:

- **Legend Toggles:** For multi-series charts (multiple lines, bars, etc.), the legend will be interactive. Users can click a legend item to hide/show that series. This is often built-in in chart libraries or can be implemented by filtering data in state. For example, if a line chart has lines for each spending category, clicking "Food" in the legend could toggle the visibility of the Food spending line.
- **On-Chart Filtering:** Users can directly click or select chart elements to filter or get more detail. For instance:
- Clicking a bar in a bar chart could filter the data to that category. In a personal finance scenario, clicking the “Food” bar might navigate the user to a detailed view or update the chart to show a further breakdown (e.g. Food expenses by month). We can either navigate to another page (drill-down page) or update the current chart to drill down. One design is to have clicking a bar add a filter context (shown above the chart as a breadcrumb). For example, the user clicks “Food” bar on an “Expenses by Category” chart, and the UI adds a filter pill “Category = Food”. Then the chart could transition to showing “Expenses over time for Food” (if that query is available or can be derived).
- Similarly, in a line chart of monthly expenses, the user might click on the point for March. We could intercept that event and perhaps show a popup with detailed transactions in March, or allow a drilldown “View daily breakdown for March” which opens a new chart or page.
- For a pie chart, clicking a slice could do something similar to the bar example – set a filter by that category and perhaps switch to a different chart focusing on that category.
- **Brush/Zoom:** For time series, enabling a brush selection to zoom into a timeframe is very useful. We could provide a lower mini-chart for range selection or allow click-and-drag to select a range. Once a range is selected, the chart could zoom into that range or trigger an updated query (if the query can accept a date range parameter). Because our saved queries might have parameters, we could integrate this: e.g., the “Transactions over Year” query might accept start and end date. The Chart Viewer could be parameterized and when the user selects Jan–Mar on the timeline, we rerun the query with that date range and update the chart. This is an advanced feature but greatly enhances analysis capability.
- **Drill-down Navigation:** In some cases, a drilldown might be a separate template or page. For example, the user clicks on a data point, and we navigate to a new route (passing some state like the filter). That new route could load another Chart Viewer configured for the more granular query. We should maintain state so the user can go back to the higher level easily. This can be done via Next.js routing (passing query params or state) or using a global state store for filters.

Implementing these interactions will involve adding event handlers to chart elements. Many React chart libraries allow attaching callbacks. For example, Recharts lets you capture clicks on bars or pie slices via props like `<Bar onClick={handleBarClick} />`. We will use such hooks to call our filter/drilldown logic. In a custom D3 implementation, we would add DOM event listeners on drawn elements. The handler will interpret what was clicked (e.g. which category or data point) and then perform the appropriate action (filter data, update state, or navigate).

State management for filters: If the app supports global filters (like "currently selected category = Food"), we might use a context or a URL query parameter to store that. The Chart Viewer could subscribe to this context so that multiple charts on a dashboard could all filter when one is selected (coordinated views). This goes beyond the question scope, but it's a consideration for expansion.

In summary, the UX will allow users not only to passively view charts but to actively customize and explore them: - They can **save templates** with a live preview feedback loop, - Maintain a **gallery** of their charts, - **Swap chart views** on the fly, - And **interact** with charts to filter data or drill into details.

All these interactions will be implemented in a user-friendly way, making heavy use of React's interactive capabilities and the chart library's features.

Data Models and Persistence Layer

We will now define the data contracts (schema) for storing charts and templates, and discuss how they fit into Firebase/Firestore and a potential Postgres migration.

Chart and Template Data Structures

Based on the specification:

- `charts` collection/table: This stores chart configurations that are tied to a specific data query (and user). Each record has:
- `id`: a unique identifier (could be an auto ID in Firestore or a UUID/serial in Postgres).
- `userId`: the user who created the chart. In Firestore, we might store this as a string (user UID).
- `queryId`: reference to the saved query or dataset this chart is for. (If using Firestore, this could be a document ID in a `queries` collection; in Postgres, a foreign key to a queries table).
- `spec`: the JSON spec of the chart (following the schema we defined). In Firestore, we can store this as a map (hierarchical fields) or as a raw JSON string. In Postgres, this would be a JSONB column containing the chart spec.
- `isDefault`: a boolean indicating if this chart is the default view for the given user & query context. At most one chart per (user, queryId) should have this true. If a chart is marked default, the app will use this spec by default when that user views the associated query. Other charts would then be considered alternate views.
- `createdAt`: timestamp of creation (and possibly we add `updatedAt` for modifications).
- `chart_templates` collection/table: This stores reusable chart templates not bound to a single dataset. Fields include:
- `id`: unique ID.
- `ownerUserId`: the user who created the template (or null/system for global templates).
- `spec`: the JSON spec of the chart template. One distinction: this spec might have a placeholder or omit `queryRef` if it's meant to be generic. In practice, we could store it with a `queryRef` that points to a sample or just leave `queryRef` blank or as a special value. When a user applies a template to a specific query, we substitute the real `queryRef`.

- `visibility`: who can see this template. Possible values: `"private"` (only the owner), `"public"` (accessible to all users globally), or perhaps `"org"` for organization-wide if multi-tenant. This field will determine authorization and listing. For instance, a template with `visibility = 'public'` will appear in every user's gallery of available templates (likely under a "Public Templates" section).

Additionally, we will have a `queries` collection/table (though not explicitly asked, it's implied since charts reference queries): - For Firestore, a `queries` collection could contain documents with query details like `{ id, userId, name, description, source, params... }`. - For Postgres, a `queries` table with similar fields. The `source` might be something like a Firestore collection name or a SQL SELECT statement, depending on implementation. Essentially, saved queries encapsulate the logic to get data (so that charts don't have to store large data or complex logic themselves).

Example Firestore documents:

- `charts/abc123: { userId: 'uid123', queryId: 'monthly_spending', spec: { ... }, isDefault: true, createdAt: ... }`
- `chart_templates/templateX: { ownerUserId: 'uid123', spec: { ... }, visibility: 'private' }`
- `chart_templates/templateY: { ownerUserId: 'adminUser', spec: { ... }, visibility: 'public' }` – a global template provided by an admin.

In Firestore, we will **secure access** to these with rules: - A user can **read** their own charts and templates. For `charts`, a rule could enforce `request.auth.uid == resource.data.userId` for reads and writes. For templates, we allow read access if `visibility == "public"` or if `request.auth.uid == resource.data.ownerUserId`. Writes only if the user is the owner (or if public, only certain roles like admin can write). This ensures users only modify their own templates and can only see others' templates if they are shared globally. For multi-tenant scenarios, we would also incorporate tenant/org checks (explained below). - We will likely **not index the entire spec field** in Firestore, as it can be large and is not needed for queries. We can mark it as an **excluded field from indexing** to save costs [8]. Firestore allows setting an exemption on a field like `spec` so it won't bloat indexes or hit size limits. We will index `userId`, `queryId`, and maybe `isDefault` so we can query for "get default chart for user X and query Y" efficiently.

Postgres schema example:

```
CREATE TABLE users (
  id UUID PRIMARY KEY,
  ... -- user fields
);
CREATE TABLE queries (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  name text,
  description text,
  -- fields defining the query (could be JSON or multiple columns depending on
```

```
implementation)
);

CREATE TABLE charts (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id),
  query_id UUID REFERENCES queries(id),
  spec JSONB NOT NULL,
  is_default BOOLEAN NOT NULL DEFAULT FALSE,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE TABLE chart_templates (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  owner_user_id UUID REFERENCES users(id),
  spec JSONB NOT NULL,
  visibility VARCHAR(10) CHECK (visibility IN ('private','public','org')) NOT
NULL DEFAULT 'private',
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

We choose JSONB for spec to easily query parts if needed (and it's indexed as a JSON binary). We might add an index on `(user_id, query_id)` for charts to find a user's charts for a specific query quickly. Also, perhaps ensure uniqueness such that each user-query pair has at most one `is_default = true` (this can be managed by app logic or a partial unique index).

With this schema, fetching a user's default chart for a query in SQL is straightforward:

```
SELECT spec
FROM charts
WHERE user_id = $currentUser
  AND query_id = $queryId
  AND is_default = TRUE;
```

If none, then use system default.

Multi-Tenancy and Authentication Considerations

The application must support multiple tenants (e.g. multiple organizations using the app, where each org's data and templates are isolated). Currently, Firebase Authentication and Firestore are used:

- **Authentication:** Firebase Auth provides `uid` for users. For multi-tenant, Firebase offers a feature where each user can belong to a tenant (using tenant IDs and tenant-aware auth). If using Firebase's multi-tenant feature, `request.auth.token.firebase.tenant` will contain the tenant ID of the user [9]. We can enforce in Firestore rules that any data read/write has a matching tenant field.

- **Firestore Data Partitioning:** We should include an `orgId` (tenant identifier) on relevant documents (queries, charts, templates). For instance, in `queries` and `charts` documents, include `orgId`. Then security rules can do:

```
match /charts/{id} {
  allow read, write: if request.auth != null
     && request.auth.token.firebase.tenant == resource.data.orgId
     && request.auth.uid == resource.data.userId;
}
match /chart_templates/{id} {
  allow read: if request.auth != null && (
       resource.data.visibility == "public" && resource.data.orgId ==
request.auth.token.firebase.tenant
    || resource.data.ownerUserId == request.auth.uid);
  allow write: if request.auth != null
     && request.auth.uid == request.resource.data.ownerUserId
     && request.auth.token.firebase.tenant == request.resource.data.orgId;
}
```

  This ensures cross-tenant access is prohibited (one org cannot see another's private templates or charts). Public templates might be either truly global (shared across all tenants) or org-specific depending on needs. If we want some templates to be shared only within an organization, we could use `visibility = "org"` and treat those similarly to public but limited to same orgId.

- **Custom Claims approach:** If not using the built-in multi-tenant feature, an alternative is to store an `orgId` as a custom claim on the user and check `request.auth.token.orgId` in rules [9]. The effect is the same – verifying that user's org matches the data's org.

On the **backend side (if migrating to Postgres)**, multi-tenancy would be enforced by application logic or row-level security: - Each relevant table (queries, charts, templates, etc.) would have an `org_id` column. - All queries from the server would include a filter on `org_id = userOrgId` (the user's org derived from their auth context). - If using an API route in Next.js, the route handler would verify the JWT or session, determine the user and org, then run a database query constrained to that org. - Alternatively, if using an ORM like Prisma, we'd include the orgId in the `where` clause for each query, or use middleware to automatically inject org filters. - For strict security, Postgres's row-level security (RLS) could be enabled: for example, set up policies such that a user with certain JWT claims can only select rows where org_id matches. If using a service like Supabase (which is Postgres-based), it provides built-in RLS using JWT claims from their auth.

Ensuring data isolation: With these measures, each tenant's charts and templates are isolated. A user from Company A cannot query or see any charts or templates from Company B. The global "public" templates could either be truly global (visible to all tenants) if they are generic, or we might decide to keep templates separate per tenant (i.e. an admin of each org can publish templates for their org). This detail can be configured via the `visibility` field or a separate field for tenant on the template.

Authentication flow: In the frontend, we will use Firebase Auth (or NextAuth etc. if migrating) to authenticate users. This gives us a user ID (and tenant ID if applicable). When making Firestore queries, security rules automatically apply as above. When calling our Next.js API (in a Postgres scenario), we include the user's auth token or session; the API will verify it and know the user and tenant, then fetch appropriate data.

Storage and Performance Considerations

Storing the chart spec JSON is not very heavy (most specs will be on the order of a few hundred to a few thousand bytes). Firestore can handle that, but as mentioned, we will turn off indexing on the spec field [8] to reduce write costs and latency. Firestore has a 1MB per document limit, which we won't hit with specs, but if a user saved an extremely complex chart with lots of annotations or something, we should still be far below that.

If any field (like `description` or a large number of annotations) could be long text, consider the indexing exemption as well.

When migrating to Postgres, the JSONB fields can be indexed with GIN if we need to query inside them (unlikely). More often, we fetch by ID or by userId+queryId, and then just use the JSON as is.

We should also consider using a **versioning** or **migration** strategy for specs. If we improve the schema later (say we add a new property or change how something is structured), we might store a `specVersion` in the JSON or as a separate field to know how to interpret it. Alternatively, ensure backward compatibility in the code when reading older specs.

Code Snippets – Saving and Loading Charts

Below are a few simplified code examples to illustrate how we might create, fetch, and enforce constraints on these data models:

Saving a new chart (Firestore example in a Next.js API route or Firebase Function):

```
// Assuming we have the spec object from the request and currentUser from auth
const newChart = {
  userId: currentUser.uid,
  queryId: req.body.queryId,
  spec: req.body.spec,            // spec should be validated against our schema
  isDefault: !!req.body.isDefault,
  createdAt: FieldValue.serverTimestamp()
};

// If isDefault is true, clear any existing default for that user+query in
Firestore:
if (newChart.isDefault) {
  const q = query(collection(db, "charts"),
                  where("userId", "==", currentUser.uid),
```

```
                where("queryId", "==", newChart.queryId),
                where("isDefault", "==", true));
  const existing = await getDocs(q);
  for (const doc of existing.docs) {
    await updateDoc(doc.ref, { isDefault: false });
  }
}
const docRef = await addDoc(collection(db, "charts"), newChart);
```

The above ensures only one default per query. (In a real implementation, you might use a transaction or batch write to avoid race conditions, but this is the concept.)

Loading a chart (deciding which spec to use):

When rendering a chart for a given user and query:

```
async function loadChartSpec(userId, queryId) {
  // First, look for a default chart for this user+query
  const q = query(collection(db, "charts"),
                where("userId","==", userId),
                where("queryId","==", queryId),
                where("isDefault","==", true));
  const snap = await getDocs(q);
  if (!snap.empty) {
    return snap.docs[0].data().spec;
  }
  // Otherwise, load system default template for this query context:
  const defaultSpec = getDefaultTemplateForQuery(queryId);
  return defaultSpec;
}
```

(This pseudo-code uses Firestore; in Postgres it would be a SELECT query. `getDefaultTemplateForQuery` could be a lookup in a config file or a query to `chart_templates` for a template marked as default for that context.)

Security Rules example (Firestore):

```
// Pseudo security rule snippet (not full syntax)
match /charts/{chartId} {
  allow read, write: if request.auth != null
    && request.auth.uid == resource.data.userId
    && request.auth.token.firebase.tenant == resource.data.orgId;
}
match /chart_templates/{templateId} {
  allow read: if request.auth != null && (
```

```
      resource.data.visibility == "public"
      || resource.data.ownerUserId == request.auth.uid
      && request.auth.token.firebase.tenant == resource.data.orgId );
  allow write: if request.auth != null
      && request.auth.uid == request.resource.data.ownerUserId
      && request.auth.token.firebase.tenant == request.resource.data.orgId;
}
```

This uses the tenant claim [9] and matches user IDs for access. It demonstrates how multi-tenant and per-user access is enforced on the backend.

In a Postgres environment, similar rules would be enforced in code or via RLS. For example, using a Node.js API, one might do:

```
app.get('/api/charts/:queryId', async (req, res) => {
  const user = await authenticate(req); // get user and org from token
  const { queryId } = req.params;
  const result = await db.query(
    'SELECT spec FROM charts WHERE user_id = $1 AND query_id = $2 AND
(is_default OR $3)',
    [user.id, queryId, true]  // perhaps fetch any default (is_default true) or
something;
    // this is simplified; we might default to system template if none found
  );
  ...
});
```

We would ensure `user.orgId` matches the query's orgId if queries are shared across orgs. Probably the query table would have orgId too to validate.

In conclusion, the persistence layer is structured to support user-specific charts and shareable templates, with careful consideration for security and multi-tenancy: - Charts tied to queries and users are stored and indexed by user and query. - Templates are stored separately for reuse, with visibility controls. - Firestore rules or Postgres logic enforce that users only access their own or shared resources. - The JSON spec is stored as-is, which provides flexibility (we don't need multiple columns for each setting) and is directly used by the frontend to render charts.

Accessibility and Design Considerations

Designing the chart viewer with accessibility (a11y) and good visual design in mind is essential. We will ensure the component adheres to accessibility standards and is usable by people with disabilities, as well as aesthetically consistent (using Tailwind for styling).

Color Scheme and Color-Blind Friendly Design

Financial charts often rely on color to distinguish categories or highlight values, so we must choose colors carefully. We will use **color-blind friendly defaults**, leveraging modern color systems like **OKLCH** color space to generate palettes with consistent contrast. By using OKLCH-based color ramps, we can ensure that colors are perceptually uniform in lightness, which helps users distinguish chart elements more easily [10]. For example, we can define a base hue and then create a series of colors by varying chroma and lightness in OKLCH, yielding a palette where each color is distinct yet balanced. This technique avoids the issue where two different hues might have very different perceived brightness (a common problem in HSL/RGB) [11] [12].

We will also integrate the **Advanced Perceptual Contrast Algorithm (APCA)** to check text and element contrast. APCA is a new contrast algorithm (part of evolving WCAG 3) that more accurately reflects human vision than the old WCAG 2.x contrast ratio [13] [14]. Using APCA, we'll ensure that chart labels, axis text, and tooltip text have sufficient contrast against backgrounds or chart colors. For instance, if we place white text on a colored bar, APCA can help determine if that white is actually readable given the hue and lightness of the bar color (sometimes APCA might say black text is better even if old WCAG would allow white). Our design system can include something like Evil Martians' Harmony palette, which was built with OKLCH and APCA to maintain consistent contrast levels [10]. By using such palettes or generating our own via OKLCH, we ensure, for example, that every series color at a given lightness level will be distinguishable when plotted together and any overlaid text or lines will meet contrast targets.

Concretely, we'll choose a default palette (perhaps 6-8 colors) optimized for color blindness – often this means avoiding confusable combinations like red/green. A common safe palette is one that uses **blue and orange/red hues** which are easier for color-blind viewers to differentiate [15] [16]. We'll avoid sole reliance on green vs red contrasts for important info (like profit vs loss should not be only green/red; we could use different shapes or additional cues). If the app's branding allows, sticking to a **single-hue palette with lightness variations** is the most robust for colorblindness [17], or using distinct hues that remain discernible (blue, orange, etc.). Black and white is the ultimate safe contrast [16], so we will ensure that if the chart were converted to grayscale, the information is still largely interpretable (e.g., by using different dash patterns or shapes for lines).

Additionally, we will incorporate **redundant encodings**: use more than just color to convey differences. For example: - Different line styles (solid, dashed, dotted) for different series, so even if two lines look similar in color, the pattern distinguishes them [18]. - Varying marker shapes for scatter plot categories. - Direct labeling on the chart where possible instead of using only a color-coded legend [19]. Placing labels next to lines or on pie slices means even if a user can't see the color well, they can read the label. - Adding **textures or stroke outlines** for filled areas: e.g. a bar could have a stripe pattern or an outline to differentiate it from another if colors clash [20].

We will test the color schemes with color blindness simulators to ensure readability, as recommended by data viz best practices [21]. These practices collectively make our charts usable for the ~8% of men and 0.5% of women who have some form of color vision deficiency [22].

Contrast and Font Sizes

All text (axis labels, tick marks, legend text, tooltips) will be styled to meet contrast guidelines. Using Tailwind CSS utility classes, we can quickly apply high-contrast colors (e.g. `text-gray-800` on a white

background for axes). APCA will guide our color choices for text over graphics. For example, if we overlay text on a colored area (like a total in big text over a shaded area chart), we ensure the shade is dark enough or choose the text color dynamically (black or white) based on background luminance. APCA differs by considering font weight/size [23]; while we might not dynamically calculate for every scenario, we'll choose conservative combinations (e.g. avoid thin light-gray text on anything but very dark backgrounds).

Tailwind's design tokens and classes will help maintain consistency. We'll define a set of CSS variables or classes for chart themes (including color scales and font sizes). We should also ensure text scales well for different screen sizes – using responsive utility classes to perhaps make chart text a bit larger on small mobile screens for readability.

Keyboard Navigation and Screen Reader Accessibility

Interactive charts can be challenging for screen readers, but we will implement basic accessibility: - The chart container will have an appropriate ARIA role, such as `role="img"` with an accessible name/label that summarizes the chart. We can use the `title` or `description` from the spec to populate an `aria-label` (e.g., `<section aria-label="Chart showing monthly spend vs budget">...</section>`). - We will provide a textual summary or table as an alternative for those who cannot perceive the chart. For example, beneath the chart or via a "Data table" toggle, we can list the key data points in text form. This ensures that even if the chart itself isn't easily navigable via screen reader, the information is available. - Focus order: If the chart has interactive elements (like clickable bars or legend items), we need to include them in the tab order. For instance, legend items could be rendered as actual buttons or list items that are focusable and have an `onKeyDown` handler to toggle series on Enter/Space key. We might wrap the SVG chart in an HTML structure that allows keyboard controls (though fully keyboard-controlling a data exploration is complex, we at least ensure the basic interactive elements respond to keyboard). - We will document keyboard shortcuts if any (like arrow keys to move along a timeline if we implement that, etc.), though initially most interactions can be translated to buttons (e.g., a "Next" button to move to next drilldown, etc., if needed).

Reduced Motion and Animations

The chart viewer will respect users' **reduced motion** preferences. Any animations (such as drawing the chart lines on load, bar growing animations, or hover enlargements) will be disabled or minimized if the user has `prefers-reduced-motion` enabled. We can detect this via CSS media queries or use Tailwind's built-in modifiers. Tailwind provides utilities like `motion-safe` and `motion-reduce` to conditionally apply animations [24]. For example, if we have an entry animation for the chart (say, bars sliding in), we would add `motion-safe:animate-slideIn` to the element, and ensure that without that (or with `motion-reduce`), it either instantly appears or uses a simpler transition.

Example Tailwind usage:

```
<div class="chart-container motion-safe:animate-fade-in motion-reduce:animate-
none">
  <!-- Chart SVG/Canvas -->
</div>
```

This would animate only for users who have not expressed a preference against motion [24]. Similarly, any auto-playing animations (like an updating live ticker) will be paused or made manual if `prefers-reduced-motion` is on.

We will also avoid unnecessary flashing or intense animations even for general users – charts will have smooth transitions but not distracting ones. Tooltips might fade in/out quickly, and updates (like when data refreshes) can be animated subtly (e.g., using D3's transition interpolation for smooth update) but we'll provide an instant update mode for reduced-motion users.

Responsive Design and Tailwind Integration

Using Tailwind CSS, we'll ensure the chart components are responsive. The chart container can be made fluid with `max-w-full` and using the chart library's responsive container (Recharts, for instance, has a `<ResponsiveContainer>` that auto-resizes the chart). We can use media queries (Tailwind's breakpoints) to adjust layout: for example, on mobile, maybe the legend positions differently (below the chart instead of to the side), or labels might be rotated/shown differently to fit.

We'll define Tailwind styles for light/dark mode as well if needed (Tailwind can swap color palettes in dark mode). The chart color palette might need a different set of colors for dark background (e.g., slightly lighter colors or a different base). This could be managed by detecting theme and loading appropriate palette (since we use CSS variables or Tailwind classes for colors).

All interactive elements will be given adequate size for touch (min 44px targets as per guidelines). For instance, legend items or buttons in the UI will have padding via Tailwind classes to be finger-friendly on mobile.

By following these accessibility and design practices, our chart viewer will be usable and clear: - **Color choices** and patterns ensure everyone (including color-blind users) can interpret the charts. - **Contrast and text** sizing ensure readability under various conditions (projector, mobile outside, etc.). - **Responsive and reduced-motion** support makes it adaptable to user preferences and devices. - **Proper semantics** and optional data tables ensure that even screen reader users or those who can't see the chart well can still get the information.

In implementing these, we are guided by known standards and tools. For example, we know that using OKLCH and APCA leads to consistent, accessible color shades [10], and adding shapes or labels in addition to color is recommended for colorblind accessibility [25]. We also directly utilize Tailwind's capability to handle reduced motion preferences [24]. This holistic approach means our charts won't just look good – they will also be **accessible to all users**.

Backend Architecture and Future Considerations

Finally, let's address how the backend supports this chart viewer, especially considering the current **Firebase/Firestore backend and a possible shift to Postgres**. We've touched on data schemas and real-time capabilities, but here we summarize and add context:

- **Current (Firestore) Implementation:** Firestore provides easy realtime and client-side access, which is great for the interactive needs of the chart viewer. We will use Firestore to store `charts` and `chart_templates` as outlined, and saved queries as well. Firestore's advantage is that the front-end (Next.js app) can directly subscribe to data. With proper security rules in place (as described), we can safely allow the client to query the needed documents. For instance, the Next.js app could use the Firebase JS SDK within a client-side component to fetch the chart spec and data. However, with Next.js, especially using **Server Components** or API routes for data fetching, we might sometimes call Firestore from the server side (to pre-render or for security). We'll likely do a mix: use server-side fetching for initial page load (for SEO or faster first paint) and then client-side for live updates. Next.js 13 with the App Router allows using React hooks in Client Components – since chart data is user-specific and not public, it's fine to fetch on the client after load.

- **Migration to Postgres:** If we migrate to a relational DB (likely with an API layer), performance and flexibility might improve for complex queries. We should design the system to **abstract data access** so that whether the data comes from Firestore or Postgres, the chart viewer usage doesn't change. For example, have a repository or service in our app that has methods like `getChartSpec(user, query)` or `runQuery(queryId, params)` which internally handle the difference between Firestore and SQL. This way, the UI code doesn't care where data is from.

- In Postgres, we might use an ORM (Prisma or Sequelize) or a GraphQL layer. We'll have to implement real-time differently: either using something like **Postgres LISTEN/NOTIFY** with websockets, or simply fall back to re-fetch on intervals for data that changes frequently (or use a service like Hasura or Supabase which gives realtime channels on database changes).

- We will also consider using **Redis or in-memory caching** for frequently accessed aggregated data, as mentioned. Postgres can handle a lot, but if we have heavy analytical queries, caching them (or pre-computing with cron jobs) is wise.

- Multi-tenancy in Postgres may involve adding an `org_id` on tables and using it in every query (ensured via middleware or within the query logic).

- **Authentication in Next.js:** If using Firebase Auth, on the server side we can verify the token to get the user. On the client, Firebase handles it. If moving to another auth (say Auth0 or NextAuth with credentials), we ensure the user context is provided to our data fetching so that we filter data accordingly.

- **Server-Side Rendering (SSR):** For initial loads, we may want to SSR the chart or at least the frame (especially if using Next.js pages that show charts in a dashboard). SSR of a chart is tricky if using a purely client-side library like Recharts (which requires DOM). One approach: use a placeholder or a loading state on SSR, then hydrate on client. Or use a headless chart renderer on server to produce

an SVG. Vega-Lite, for instance, can be used server-side to generate an SVG/PNG from a spec. We might not need that initially. We can mark the chart component as a client component (`"use client"` at top) since most chart libraries are not compatible with Next.js Server Components [26]. This is fine, just means the chart will load after hydration. We can show a spinner or skeleton in the meantime.

- **Scaling Considerations:** As more users and data come in, Firestore's limitation on querying (like it can't do complex aggregations or text search easily) might push us to move to Postgres for those features. The design we created stores specs and templates in a straightforward way that translates well to either DB. The bigger challenge is the data for the charts (the saved queries results). We might end up moving transaction data into Postgres for robust querying. At that point, saved queries might become actual SQL views or just queries run on the fly. We should ensure the chart viewer can get data from either source. Possibly we adopt a strategy where for now, we use Firestore for everything, and gradually migrate certain queries to an interim API that queries BigQuery or Postgres for heavy analysis. For example, for a very heavy report, we could export data to BigQuery and run a query there, returning the result to the app.

- **Testing and Error Handling:** We will implement thorough testing for the chart component and its data loading. This includes unit tests for the spec parsing (ensuring that given a spec and sample data, the correct chart config is produced), and integration tests for the saving/applying template flows. Also, proper error handling: if a query fails to load (maybe network issue or auth issue), the chart component should display a friendly error message ("Failed to load data") instead of just breaking. If a user attempts to apply a template that doesn't match the data, we catch that and maybe show a warning like "This template isn't compatible with this dataset."

By addressing backend context in our implementation guide, we ensure the solution is viable both now and in the future state: - The design is cloud-agnostic enough (works with Firestore now, but structured enough to map to SQL later). - Security and multi-tenancy are baked in via either Firestore rules or application logic, maintaining **data isolation and proper auth** [9]. - We leverage Firestore's real-time strengths now, and plan for alternative solutions when moving to Postgres (perhaps using libraries or services that provide real-time subscription to changes, or simply adjusting user expectations for what needs to be real-time).

Conclusion: This implementation guide outlined how to build a comprehensive interactive chart viewer in a Next.js/React application with Tailwind and TypeScript. We defined a flexible JSON schema for chart specifications that enables a robust templating system, integrated with user-saved queries for live data. We detailed how the user experience allows creating, saving, and switching between chart templates with live previews and interactive drilldowns. We specified data models for storing charts and templates, along with strategies for caching and real-time updates to keep performance smooth. We also emphasized accessibility (color choices, contrast, reduced motion) and how to implement those with modern tools and standards [10] [24]. Finally, we connected these plans with backend realities in Firebase and a potential Postgres environment, ensuring the solution is secure, multi-tenant capable, and scalable. With this plan, developers can implement the chart viewer step by step – from defining the JSON schema and building React components, to setting up Firestore rules or database tables – resulting in a feature-rich personal finance charting tool that is both **reusable** and **user-customizable**.

Sources:

- Recharts library overview – demonstrating support for numerous chart types and React integration [1].
- Holistics documentation on custom chart templates with Vega-Lite – inspiration for user-defined JSON chart specs and org-wide sharing [3].
- Evil Martians' Harmony palette – using OKLCH color space and APCA for accessible color design [10].
- APCA contrast algorithm explanation – benefits over old contrast methods for readability [14].
- Datylon guide on colorblind-friendly visualization – techniques like using shapes, direct labels, safe color palettes [25] [16].
- Tailwind CSS docs – using `motion-safe` / `motion-reduce` for respecting reduced motion preferences [24].
- Firebase Firestore best practices – advising to exempt large unqueried fields (like our JSON specs) from indexing [8].
- StackOverflow example – securing multi-tenant Firestore data by matching auth token tenant ID to data's tenant field [9].

---

[1] [2] [7] [26] How to use Next.js and Recharts to build an information dashboard
https://ably.com/blog/informational-dashboard-with-nextjs-and-recharts

[3] Tutorial: Create a Custom Chart from Vega-Lite library | Holistics Docs (4.0)
https://docs.holistics.io/guides/create-custom-chart-from-vega-lite-lib

[4] Load Data Faster and Lower Your Costs with Firestore Data Bundles!
https://firebase.blog/posts/2021/04/firestore-supports-data-bundles/

[5] [6] Write-time aggregations | Firestore | Firebase
https://firebase.google.com/docs/firestore/solutions/aggregation

[8] Best practices for Cloud Firestore | Firebase
https://firebase.google.com/docs/firestore/best-practices

[9] firebase - Firestore Rules with multi-tenancy? - Stack Overflow
https://stackoverflow.com/questions/63291425/firestore-rules-with-multi-tenancy

[10] GitHub - evilmartians/harmony: Harmony color palette
https://github.com/evilmartians/harmony

[11] [12] Color experiments with OKLCH – Chris Henrick
https://clhenrick.io/blog/color-experiments-with-oklch/

[13] [14] [23] APCA: the new algorithm for accessible colour contrast · Juan Ruitiña
https://ruitina.com/apca-accessible-colour-contrast/

[15] [16] [17] [18] [19] [20] [21] [22] [25] The best charts for color blind viewers | Blog | Datylon
https://www.datylon.com/blog/data-visualization-for-colorblind-readers

[24] animation - Transitions & Animation - Tailwind CSS
https://tailwindcss.com/docs/animation