ChatGPT

Provider-Agnostic Economic Indicator Subsystem – Design & Implementation Guide

Overview and Objectives

A robust economic indicator subsystem is crucial for a personal finance app to incorporate live economic context into user finances. This guide outlines a provider-agnostic subsystem that acquires, normalizes, caches, and binds economic indicators (initially focused on Brazil, with a design extensible globally). Key indicators include Brazil's inflation index (IPCA), benchmark interest rates (SELIC, CDI), foreign exchange rates (e.g. USD/BRL), and other domain-specific indices [1]. The system is designed to seamlessly switch data providers, ensure data integrity, and integrate these indicators into various app features (reports, budgets, forecasts, etc.).

Goals of the subsystem:

- **Flexible Indicator Catalog:** Maintain a schema of all supported indicators with metadata like frequency (daily, monthly), units (percent, index points, currency), base/reference (for index baseline), seasonal adjustment, and preferred data sources.
- **Provider Abstraction & Reliability:** Fetch data from primary sources (e.g. Central Bank APIs) with fallback to secondary or cached sources. Enforce caching (e.g. ~24h TTL) to reduce load and support offline snapshots. Detect anomalies (e.g. non-monotonic index or extreme spikes) and handle backfilling of historical data.
- **Normalization & Transformations:** Provide utilities to convert values between **nominal and real terms** (inflation adjustments), perform **discounting** using interest rate curves (SELIC/CDI), compute forward rates, and handle **FX conversions** (with official daily fix vs intraday market rates).
- **Application Binding:** Cleanly integrate indicators into app features – e.g. toggling reports between real vs nominal values, automatically inflation-adjusting budget projections, using indicators as exogenous variables in forecasts, and enabling scenario modeling (optimistic/pessimistic cases, ±σ variations).
- **Data Contracts & Versioning:** Define clear data contracts for indicator values, groupings of indicators (sets), and binding configurations. Include an `asOf` **timestamp** on data to support versioning for reproducibility/backtesting (ability to use vintage data as known on a given date).
- **Reliability & Ops:** Implement health checks and alerts for missing or stale data, re-compute dependent forecasts on data revision, and maintain an audit trail of which indicator values were used in calculations.

Throughout this guide, we provide TypeScript interface definitions, code snippets, and examples to illustrate the implementation. We also include a QA checklist and testing strategy (using golden files and snapshot tests) to ensure the subsystem's correctness and stability over time.

Indicator Catalog & Units

Indicator Catalog Schema: We maintain a master catalog of all economic indicators supported by the app, each defined with key metadata. This catalog drives how data is interpreted and processed. The schema includes:

- `code` – A unique identifier (string) for the indicator (e.g. `"IPCA"`, `"SELIC"`, `"USD/BRL"`).
- `name` – Human-readable name/description (e.g. *"IPCA – Broad Consumer Price Index"*).
- `frequency` – Granularity of the time series, such as `"daily"`, `"monthly"`, `"annual"`, etc.
- `unit` – Unit of the values, e.g. `"index"` (points), `"%"` (percentage rate), or currency code (`"BRL"`, `"USD"` for FX rates).
- `base` – Reference base for index series or rates. For index values, this could be a base year or value (e.g. IPCA index base = 100 in a reference year). For rates, this might indicate day-count convention (252 business days for Brazil's CDI/SELIC vs 365).
- `seasonal_adjusted` – Boolean indicating if the series is seasonally adjusted (relevant for some indices like GDP, though many finance indicators like IPCA are not seasonally adjusted).
- `source_priority` – Ordered list of data providers for this indicator. The first provider is primary; subsequent entries are secondary or fallback sources if the primary is unavailable or out-of-date.

For example, an entry might be:

```
interface IndicatorMeta {
  code: string;               // e.g. "IPCA"
  name: string;               // e.g. "IPCA – Broad Consumer Price Index"
  frequency: 'monthly' | 'daily';
  unit: 'index' | '%' | 'BRL' | 'USD' | string;
  base: string | number;      // e.g. "=100 (Dec 1993)" for IPCA index base, or
252 for CDI day count
  seasonal_adjusted: boolean;
  source_priority: string[];  // e.g. ["BCB_SGS", "OFFLINE_SNAPSHOT"]
}
```

The catalog allows the system to understand how to handle each indicator. For instance, an indicator with `frequency: "monthly"` will be stored and queried at monthly intervals, but if daily values are needed, the system can **interpolate or forward-fill** using the last monthly value (according to defined interpolation rules).

Daily vs Monthly Storage: The subsystem's storage (e.g. database or in-memory cache) must accommodate different frequencies. A common approach is storing all series in a unified time-series table with a date field. For monthly data, the convention can be to use the first day of the month or a consistent representative date (like last day) for that period. The system should clearly document how it represents period dates. When bridging frequencies, simple rules apply – e.g. if an annual value is requested in the middle of the year, use the latest available annual value (or perhaps linearly interpolate if appropriate).

Interpolation Rules: For compatibility across frequencies, define how to infer values for dates between official data points: - **Forward-fill (Step interpolation):** Appropriate for many economic indices – assume

the latest known value holds until an update. For example, a monthly CPI index could be treated as constant throughout the month until next release. - **Linear interpolation:** Could be used for smoother transitions if needed (less common for economic indicators, but sometimes used for mid-period estimates). - **Business-day handling:** For daily series that only have values on business days (e.g. market rates), the system might choose to carry forward the last value for non-trading days (weekends/holidays) if a continuous daily series is needed.

Revision History & Versioning: Some indicators (especially economic statistics) get revised after initial release. To support reproducibility and backtesting, the storage should maintain **versions** of data points. This is handled via an `asOf` **timestamp** on each data entry: - The `asOf` represents when that data value was known/released. If an indicator's past value is updated later, a new record is added with a newer `asOf`. The system **does not overwrite** past values; instead it keeps multiple records (time series versions). - This allows retrieval of "vintage" data – data as it was available on a prior date. For example, the U.S. ALFRED system (by the St. Louis Fed) archives vintage economic data for exactly this reason, enabling users to gather data **as reported on past dates** to reproduce historical analyses [2]. We apply the same principle: you can query the indicator value as of a specific cut-off date for backtesting models with only information that existed then.

A TypeScript type for stored indicator values might be:

```
interface IndicatorValue {
  code: string;
  date: string;       // e.g. "2025-08-01"
  value: number;
  unit: string;       // e.g. "%" or "BRL"
  asOf: string;       // ISO timestamp of data release or fetch
  source: string;     // provider identifier (e.g. "BCB_SGS")
}
```

This design ensures we know *which* source provided the data and when. When updating values, the system compares new data to what's stored: - If new `value` differs from the latest stored value (for the same `date` and `code`), it adds a new record with a newer `asOf`. The older value remains in the history. - If values are unchanged, it can update an internal fetch timestamp without duplicating data (or simply leave data as is but update a cache timestamp).

Example: Suppose the September 2025 IPCA inflation index is first released on Oct 10, 2025 as 8120.45 (index points). We store that with `asOf = 2025-10-10`. Later, in Nov 2025, the agency revises September's value to 8120.60. We then store a new `IndicatorValue` for code `IPCA`, `date=2025-09-01`, `value=8120.60`, with `asOf = 2025-11-15` (for example). Both versions remain in the database. By default, queries will fetch the latest (max asOf) value unless a specific historical `asOf` is requested.

Providers & Fetch Strategy

Economic data can come from various providers (government statistical APIs, central banks, financial markets, etc.). A **provider-agnostic** design uses an abstraction layer to interface with any data source in a uniform way. Key aspects of the fetch strategy include primary/secondary fallback logic, caching, backfilling, anomaly detection, and consistent versioning.

Provider Abstraction Layer: We define a common interface that all data providers must implement. For example:

```
interface IndicatorProvider {
  name: string;
  /** Fetch data for an indicator code between the given dates (inclusive). */
  fetchData(code: string, startDate: string, endDate: string):
Promise<IndicatorValue[]>;
}
```

Each provider knows how to retrieve data for certain indicator codes. In configuration, each indicator's `source_priority` lists provider names in order of preference. For example, for `USD/BRL` exchange rate, `source_priority: ["BCB", "ForexAPI"]` might indicate: first try the Central Bank of Brazil (BCB) API, but if it's unavailable, use a secondary Forex API.

Primary, Secondary, Offline Sources: Providers can be of various types: - **Primary online APIs:** e.g. BCB's SGS JSON API for Brazilian economic series, which provides official data for IPCA, Selic, CDI, etc [1]. Another example for global expansion might be the U.S. FRED API for Fed economic data. - **Secondary APIs or data feeds:** alternative sources if primary fails (e.g. an FX rate from a different service if the central bank site is down). - **Offline or snapshot source:** a local cache or bundled dataset. This can serve two purposes: (1) act as a *cache layer* to avoid frequent calls, and (2) allow basic functionality when the user is offline or the provider is unreachable. For instance, the app may include a snapshot of last known values or a compressed history for critical series, so that in absence of internet it can still show some data (marked as possibly stale).

Each provider implementation handles the peculiarities of its source (endpoints, authentication, data format) and transforms data into the unified `IndicatorValue` format. For example, a BCB provider might call `https://api.bcb.gov.br/dados/serie/bcdata.sgs.{code}/dados?formato=json` and map the JSON fields to our structure. A file-based provider might load values from a local JSON/CSV.

Fetch Logic with Fallback and Caching: When the system needs data (say to update an indicator or answer a query): 1. **Check Cache:** If the data is recent enough in our cache/storage, use it. Each indicator could have a cache timestamp of last successful update. If that is within a defined TTL (time-to-live), we trust our stored data. For example, for daily series, we might set TTL = 24 hours (i.e. don't refetch more than once per day) since these values typically update daily. For intraday needs (like live FX), TTL might be shorter (e.g. 1 hour or less), but for simplicity we aim to avoid any refresh more frequent than necessary. 2. **Primary Provider Attempt:** If cache is stale or missing data for the requested period, invoke the primary provider's `fetchData`. This could fetch the latest values or a date range. 3. **Fallback on Failure:** If the primary fetch

fails (network error, or returns an error from API), the subsystem logs a warning and tries the next provider in the `source_priority` list. This chain continues until either data is obtained or all sources fail. 4. **Data Validation & Anomaly Check:** Upon receiving data, apply basic validations: - **Completeness:** Did we get data for all expected dates in range? If not, and if the provider supports historical queries, we might attempt a backfill for missing dates or mark them as missing. - **Monotonicity & Range Checks:** Some series have expected patterns. For example, a cumulative price index like IPCA tends to grow over time (deflation is possible but rare and small). If we detect a *negative* jump far beyond typical variation or a sudden spike (e.g. an interest rate jumping from 5% to 500% due to a faulty decimal), flag it. The system could either reject the data or accept but raise an alert for manual review. Anomaly detection can be as simple as threshold rules or z-score outlier detection on percentage changes. - **Unit Consistency:** Ensure the values make sense given the unit (e.g. a percentage should generally be 0–100 for rates, an index value shouldn't be negative). 5. **Caching Results:** Write the new data to the local storage, including setting the `asOf` (typically `asOf = now()` or the timestamp given by provider if any) for each record. Update the cache timestamp for that indicator. 6. **Backfill Policy:** If the request was for historical data (e.g. user scrolls a chart to past years) and our storage doesn't have it, the system can trigger a backfill: call the provider for older ranges. This can either happen on demand or lazily in background. For example, if our system initially only stored the last 2 years of a series and the user requests 5-year view, the subsystem fetches the missing 3 years from the API (if available). Backfill can also be proactive – e.g. on first setup of an indicator, fetch the full history once. 7. **Rate Limiting & Retry:** Providers might have call limits. The fetch logic should include minimal retries on transient failures and respect provider usage guidelines. For instance, using exponential backoff for retries and avoiding tight loops.

A simplified fetch with retry (pseudo-code):

```
async function fetchWithRetry(provider: IndicatorProvider, code: string, start:
string, end: string): Promise<IndicatorValue[] | null> {
  const maxAttempts = 3;
  for (let attempt = 1; attempt <= maxAttempts; attempt++) {
    try {
      const data = await provider.fetchData(code, start, end);
      return data;
    } catch (err) {
      console.warn(`Fetch failed from ${provider.name} (attempt ${attempt}): $
{err}`);
      if (attempt < maxAttempts) await delay(1000 * attempt);  // incremental
backoff
    }
  }
  return null;
}
```

Then the overall get-data flow:

```
async function getIndicatorData(code: string, start: string, end: string):
Promise<IndicatorValue[]> {
```

```
  const meta = catalog[code];
  // 1. Check cache freshness
  if (isCachedFresh(code, end)) {
    return queryLocalData(code, start, end);
  }
  // 2. Loop through providers in priority order
  for (const providerName of meta.source_priority) {
    const provider = providers[providerName];
    const result = await fetchWithRetry(provider, code, start, end);
    if (result) {
      storeData(result);  // save to DB/cache with asOf timestamps
      return queryLocalData(code, start, end);  // return unified result from
local store
    }
  }
  // 3. If all providers failed:
  const offlineData = queryLocalData(code, start, end);
  if (offlineData.length) {
    console.error(`All providers failed; returning possibly stale cached data
for ${code}.`);
    return offlineData;
  }
  throw new Error(`Unable to retrieve data for ${code} from any provider.`);
}
```

In the above: - `isCachedFresh` checks if the latest date we have for `code` is today (or within TTL) for daily series, or within the current period for lower frequency series. - `queryLocalData` pulls data from the local time-series storage. - `storeData` writes new values (with versioning as needed).

Caching and TTL: By default, we set a **TTL of 24 hours** for most daily data. That ensures we fetch at most once per day per indicator, unless the user explicitly forces an update (e.g. a “refresh” action) or if an intraday update is critical. For sub-daily updates (like intraday FX rates), we could shorten TTL or even fetch on demand without caching if the user opens a currency conversion feature – but even then, caching for a few minutes or an hour can avoid excessive calls.

For **monthly or quarterly data**, TTL can be much larger – we might check for new data only around the expected release date. E.g., IPCA (monthly) usually releases around the 10th of each month for the previous month; the system can mark the next expected update and only fetch after that date passes.

Anomaly Detection Examples: - If a supposedly monotonic index (like a cumulative price index) decreases by more than a tiny fraction, flag it. Minor deflation (a few basis points) might be legitimate, but a 5% drop in a price index would be suspicious. - If an interest rate jumps or drops by, say, >5 percentage points in a day outside of a known policy meeting outcome, that could be a data error. - The system could implement these as simple checks on the fetched `result` before storing. If anomaly detected, it could either discard the new data (keeping old) and alert developers, or store it but mark an `anomaly` flag for review. In either case, alerting (via logs or notifications) is important to ensure data reliability.

`asOf` **Versioning in Fetch:** When storing new data, the `asOf` field is typically the fetch time or the data release time. Some providers include a release timestamp. If available, we use that (for precision); otherwise, we use current time as a proxy for "first seen at". This ensures each distinct fetch cycle that yields changed data is captured as a new vintage. Consumers of data can specify an `asOf` date to retrieve historical vintages if needed (for example, to simulate running a forecast model with last month's data only, pass `asOf = <end of last month>` to getIndicatorData).

Reproducibility and Backtesting: Using `asOf` versioning, the app can support a "time machine" mode for data. This is analogous to how ALFRED or other archival systems work, allowing analysts to **reproduce past results with the data available at that time** [2]. For instance, if a user wants to backtest their budget rule with data as it was before a major revision, the system can fetch indicator values with `asOf` prior to that revision date. Internally, this would query the latest `IndicatorValue` records whose `asOf` is <= that cutoff.

Normalization & Transformations

Raw indicator data often needs to be transformed or normalized for use in calculations. This subsystem provides utility functions for common transformations: adjusting for inflation (nominal ↔ real values), interest rate conversions/discounting, forward rate calculations, and currency conversions. All transformations use the stored time-series data as inputs, ensuring consistency.

Inflation Adjustments (Nominal ↔ Real)

Context: Nominal prices or amounts at different dates are not directly comparable due to inflation. For meaningful comparisons or aggregations over time, values should be converted to **real terms** (inflation-adjusted to a base period's purchasing power). Conversely, to project nominal future values, one might *inflate* a real baseline by expected inflation rates.

Approach: Use an appropriate price index (e.g. IPCA for Brazil) to adjust values. The formula typically is:

- To express an amount from date *t* in terms of base date *b* (both dates in the past or present, *b* being the desired reference):
  Real_b(Value_t) = Value_t × (Index_b / Index_t).
  This scales the nominal value at *t* by the ratio of the price index at the base date over the index at date *t*. If *b* is today, this yields the value in "today's money".

- If instead we have a known real value and want to get the nominal value at a future date *f*:
  Nominal_f = Real_b × (Index_f / Index_b).

In implementation, we retrieve the index series (e.g., IPCA monthly index) and perform the multiplication or division. Using the ratio of index values automatically accounts for cumulative inflation between the two dates.

Example: Suppose IPCA index for Jan 2020 = 100, and for Jan 2025 = 150 (meaning 50% cumulative inflation from 2020 to 2025). A nominal price of R\$2000 in Jan 2020 expressed in Jan 2025 money would be

`2000 × (150/100) = 3000` (i.e. R\$3000 in 2025 terms). Conversely, R\$3000 in 2025 corresponds to R\$2000 of 2020 purchasing power when deflated.

Implementation – Inflation Utility: We add a function such as:

```
function adjustForInflation(amount: number, fromDate: string, toDate: string,
indexCode: string = "IPCA"): number {
  const series = queryLocalData(indexCode, fromDate, toDate);
  if (series.length === 0) throw new Error("Index data not available");
  const idx_from = series.find(v => v.date === fromDate)?.value;
  const idx_to = series.find(v => v.date === toDate)?.value;
  if (!idx_from || !idx_to) throw new Error("Index values missing for given
dates");
  return amount * (idx_to / idx_from);
}
```

This assumes `series` contains index values for at least those two dates. In practice, one might need to get the nearest available index to `fromDate` and `toDate` (e.g. if dates are not exact index release dates, pick the latest prior index for fromDate and latest prior for toDate).

Inflation Data Consideration: The IPCA (or any CPI) might be released monthly. If adjusting on a day that isn't exactly month-end, the convention could be to use the latest published index. That is acceptable given inflation doesn't usually move drastically day-to-day. Alternatively, one could interpolate a daily index (some advanced methodologies create a daily CPI proxy [3], but this is usually unnecessary for personal finance granularity).

Why Real vs Nominal Matters: Nominal values can mislead analyses of growth or comparisons over time [4]. Our app will allow users to toggle between viewing reports in nominal terms or real terms. Under the hood, this function will be used to convert values when the "real" toggle is on. As one author notes, "nominal prices over time often mislead … over-time comparisons should use 'real' (inflation-adjusted) values under most circumstances" [5] [6]. Thus, providing a reliable inflation adjustment function is fundamental.

Interest Rate Discounting and Forward Rates

Context: Interest rates come in various forms (annualized, daily, simple, compound). In Brazil, SELIC and CDI are key benchmarks. SELIC is the policy rate (annual target), and CDI is the effective daily interbank rate which closely tracks SELIC. Financial calculations often require: - Converting an annual rate to a daily rate (for accrual or discounting cash flows day by day). - Computing the accumulation of interest over a period. - Discounting a future value back to present using an interest curve. - Calculating forward rates (the implied future interest between two time periods, given a term structure of rates).

Daily vs Annual Rates (Brazilian conventions): Brazil uses 252 business days convention for annual-to-daily conversion (since interest is earned only on business days). The relationship is:
(1 + annual_rate)^(1/252) - 1 = daily_rate.
This formula yields the daily interest rate *d* corresponding to an annual rate *a* [7]. For example, if CDI is

12% p.a., the daily rate is (1+0.12)^(1/252) - 1 ≈ 0.000451 (0.0451%). This daily rate applied each business day will compound to ~12% over a year.

Using the above, a general function for converting an annual rate to a period rate given `n` compounding periods per year is:
periodic_rate = (1 + annual_rate)^(1/n) - 1.
For daily (business day) in BRL context, n = 252. For daily in a 365-day context (e.g., some global uses), n = 365.

Accumulating Interest (compound growth): To get the factor over multiple days, you multiply daily factors. If we have a series of daily rates r1, r2, ..., rn (where each r is small, like 0.0004), the accumulation over that period is:
Accumulation factor = ∏(1 + r_i) over all days in the period.
The result minus 1 gives total effective growth. In practice, for constant rate or when using the formula, one can also do: (1 + a)^(days/n) - 1 for a constant annual rate over "days" length.

Discounting: To discount a future value back to present, take the inverse of the accumulation factor for that period. For example, to discount R\$1000 one year at 12% annual (assuming compound): present value = 1000 / (1+0.12) = 892.86. If doing it by days, you'd divide by the product of (1+r_i) for each day from now until the future date.

Forward Rates: If we have an interest rate for 0–1 year and for 0–2 years, the forward rate for year 2 (i.e., the implied rate for the period between year 1 and year 2) can be derived:
(1 + r_{0-2yr})^2 / (1 + r_{0-1yr})^1 - 1 = forward_1yr rate for year2.
Generalizing: forward rate for period (t1,t2) given zero/period rates for 0–t1 and 0–t2. The subsystem can implement utilities to compute forward curves if needed for scenario analysis.

Implementation – Interest Utilities: Some examples in TypeScript:

```
const DAYS_BR_BUSINESS = 252;

function annualToDailyRate(annualRate: number, businessDays: boolean = true):
number {
  const n = businessDays ? DAYS_BR_BUSINESS : 365;
  return Math.pow(1 + annualRate, 1/n) - 1;
}

function accumulateRateDaily(dailyRates: number[]): number {
  // returns accumulated factor (1+total_return)
  return dailyRates.reduce((acc, r) => acc * (1 + r), 1);
}

// Example: discount future value with daily rate series or constant annual
function discountFutureValue(futureValue: number, annualRate: number, days:
number): number {
  const daily = annualToDailyRate(annualRate);
```

```
  const factor = Math.pow(1 + daily, days);
  return futureValue / factor;
}

function impliedForwardRate(r0T: number, r0t: number, T: number, t: number):
number {
  // r0T: zero/annual rate for 0-T period, r0t: for 0-t period, both in years
  // T > t, returns annual forward rate for period t to T
  const totalFactor = Math.pow(1 + r0T, T);
  const firstFactor = Math.pow(1 + r0t, t);
  const forwardFactor = totalFactor / firstFactor;
  const forwardAnnual = Math.pow(forwardFactor, 1/(T - t)) - 1;
  return forwardAnnual;
}
```

In practice, our indicator subsystem will likely store daily CDI rates as a time series already (the *CDI diária*, which is an overnight rate published each business day). That series itself represents the accumulation factor day by day. For example, a data point might be 0.000451 (which is the **daily yield** fraction, or sometimes they give an index factor like 1.000451). If we store the rate as a percentage or fraction, we must be careful in using it: - We might store CDI as *annualized %* (12% etc) each day. Then to use it for accrual we must convert to daily via formula each time. - Alternatively, store the **daily factor** directly (e.g., 1.000451). In that case, accumulating is just multiplying those factors.

The design choice could be to store interest rates in annual terms or period terms. Storing as annual rate is intuitive for users viewing it, but for calculation storing as factors is easier. We might choose to store both as separate indicators (e.g., `CDI_RATE` = 12% p.a., and `CDI_FACTOR` = 1+daily_rate). The `unit` field can clarify ("%" vs none). The transformation utilities can derive one from the other.

Example Use: If the user's portfolio or loan calculations require discounting cash flows with CDI: - The system fetches the daily CDI rate series for the needed range. - Compute the accumulation factor over the interval. - Apply to amount (for growth) or invert (for present value).

This is implemented behind the scenes in features like showing *"If you invest X at CDI for N days, you'll get Y"*, or updating loan balances with daily interest.

Foreign Exchange (FX) Conversion

Context: The app may support multi-currency assets or transactions, especially since global expansion is planned. Converting between currencies requires up-to-date exchange rates. Key considerations: - Using official daily reference rates (e.g. PTAX from Banco Central do Brasil for USD/BRL closing rate) vs. live market rates. - Handling intraday scenarios: if user checks during market hours, a more recent rate might be desired; after hours or weekends, last official close may be used. - Fallback: if the primary source (official rate) is delayed or missing, use an alternate feed.

Data Handling: We treat each currency pair as an indicator series, e.g. code `"USD/BRL"` with daily frequency, unit = "BRL" (value meaning 1 USD equals X BRL). We might also have `"EUR/BRL"`, etc. These usually update each business day (some feeds might have weekends as repeats or no data on holidays).

Providers: For Brazil, the Central Bank publishes an official closing rate (PTAX) each business day. For intraday or other currencies, we might use a provider like an FX data API or an open source feed. The provider abstraction allows adding those easily.

Conversion Process: To convert an amount from currency A to B on a given date: 1. Determine the indicator code for the pair (if our system always stores X/Y as value of 1 X in Y, we need to pick the right one). 2. Fetch the rate for that date (or nearest available prior date if none that day). 3. Multiply/divide accordingly: - If converting from X to Y and we have indicator X/Y (Y per 1 X): `result = amount * rate_X/Y`. - If we only have the inverse stored, use 1/rate.

Intraday and Live Rates: The subsystem can handle intraday in two ways: - *Near-real-time feed:* If a provider supplies up-to-the-minute quotes (likely not the central bank, but a market API), we can fetch that on demand. These would not be stored as historical official data (since they fluctuate); possibly we only use them for immediate conversion display. We might still wrap it in the provider interface but maybe mark them as "transient" data not to be kept as official history. - *Last known fallback:* If it's a weekend or after hours, the last closing rate (last business day) is used. The UI might indicate it's last close. Our caching logic would consider the data "fresh" until next trading day.

FX Fallback Example: If BCB's rate is unavailable today (perhaps a holiday or API down), the secondary provider might be an international source's closing rate. Or we might use yesterday's rate and flag that it's potentially stale. Given FX can move, a stale rate is not ideal, so alerting might be warranted if we go beyond 1 business day without update.

Implementation – FX Utility:

```
function convertCurrency(amount: number, from: string, to: string, date:
string): number {
  if (from === to) return amount;
  // Assume we store direct pair rates in one direction, e.g. USD/BRL
  const code = `${from}/${to}`;
  let rateRec = queryLocalData(code, date, date)[0];
  if (!rateRec) {
    // try inverse if direct not stored
    const invCode = `${to}/${from}`;
    rateRec = queryLocalData(invCode, date, date)[0];
    if (!rateRec) throw new Error("Exchange rate data not available");
    return amount / rateRec.value;
  }
  return amount * rateRec.value;
}
```

This simplistic function queries the stored rate for that date. In practice, for same-day near real-time, we might call the provider instead of `queryLocalData` if we know the market is open and we want the latest tick. The strategy could be: - If `date` = today and during trading hours, use a live provider (bypassing cache TTL). - Otherwise, use cached official daily.

All currency conversions in the app (e.g. showing a portfolio total in BRL when it has USD stocks) would use this utility under the hood. The bindings (described next) will link the UI to this function, so that whenever conversion is needed, the appropriate rate is fetched and applied.

Other Transformations

Other domain-specific transformations may include: - **Seasonal Adjustment toggling:** If we had some indicators with both seasonally adjusted and not, we could allow switching. (This might be more relevant for macro indicators like GDP, which are outside personal finance scope for now. The schema field `seasonal_adjusted` is mainly informational or for choosing default series). - **Index Re-basing:** Converting an index to a different base year or scaling (usually presentation concern). E.g., rebase an index to 100 at a user-specified date. This is a simple transformation: `new_value = old_value / old_index_at_base * 100`. - **Percentage Change Calculations:** Though not exactly an "indicator transformation", computing percent changes, year-over-year inflation, etc., could be considered part of normalization for displaying growth rates. These formulas are straightforward (e.g., (x_t/x_{t-12} - 1)*100 for YoY percent change [8]).

Binding to Application Features

With the indicator subsystem supplying cleaned and normalized data, the next step is binding these indicators into the personal finance application's features. This ensures that changes in economic data automatically reflect in user-facing functionalities. We outline how each major feature can utilize indicators:

Real vs Nominal Toggles in Reports

Many financial reports (net worth over time, expense trends, investment growth) can be viewed in nominal terms (actual figures at the time) or real terms (adjusted for inflation to remove price level changes). We provide a **toggle** for users to switch between these views.

Implementation: When the user selects "Real (inflation-adjusted)": - Determine a base date for real terms (commonly the latest date in the report or a specific base year). For example, if viewing a 5-year expense chart, base = the most recent month in the chart. - For each monetary value at date *t* in the report, use the inflation adjustment utility: `real_value = nominal_value * (Index_base / Index_t)` for the chosen price index. In Brazil, IPCA or another broad index is used as default. - The report line or bar values are replaced with these real values. The UI should label it as "values in YYYY prices" to clarify.

When the toggle is "Nominal", no adjustment is applied (or equivalently Index_base = Index_t, so values remain the same).

Binding: We can create a binding configuration that links report data to the inflation index: - E.g., `bindings: { indicator: "IPCA", target: "Reports.InflationIndex", transform: "deflate" }`. - The app's report generator will check if a report is marked to use a real-value transform. If so, it looks up the bound indicator series (IPCA) and applies the deflate transform to all values.

This binding means whenever IPCA is updated, any open report in "real" mode could also update (if it covers the latest period). However, in practice, reports likely use whatever data was last fetched until refreshed, which is fine because IPCA moves slowly (monthly).

The net effect for the user: they can see, for example, that although their nominal spending increased 10% over 3 years, in real terms it only increased 2% because inflation was ~8% in that period. This makes goals and trends more transparent, fulfilling the need to "be honest" about changes by using real values [9].

Inflation Uplifts in Budgets & Planning

When users set budgets or plan future expenses, the system can automatically **uplift** future values by expected inflation to preserve real purchasing power. For instance, a user's monthly grocery budget of R\$1000 today might be suggested as R\$1,100 next year if 10% total inflation is expected by then.

Implementation: The planning module will have access to forecasted inflation rates (could use the central bank's outlook or a simple assumption based on current rate). The subsystem might treat forecasted indicators similarly to actual indicators (perhaps under an "expected IPCA" series, which could be a static percentage or a curve of future index levels).

For each budget item or category that is marked as "auto-inflate", the system links it to the inflation indicator: - Binding example: `{ indicator: "IPCA", target: "Budget.Food", transform: "applyGrowth" }`. The `applyGrowth` transform means: take the base value and multiply by the inflation growth for the budget period. - If the budget is yearly, use year-over-year inflation forecast; if monthly, could apply monthly inflation iteratively.

In practice: When showing next year's budget in the UI, the app does:

```
nextYearValue = currentValue * (1 + expectedInflationRate);
```

or if we have an index projection:

```
nextYearValue = currentValue * (Index_{endOfNextYear} / Index_{now});
```

(using forecasted index values).

The subsystem provides `getForecastIndicator(indicator, futureDate)` which might either retrieve from a stored projection series or compute using a simple model (e.g. constant rate).

User control: The app could allow toggling this feature on/off per budget. If off, the value stays fixed (nominal). If on, it auto-updates when inflation expectations change.

Whenever the inflation forecast series is updated (say the user updates an assumption or a new official forecast is released), a re-computation of all bound budget values can be triggered (or at least a notification that "Budgets adjusted by inflation have been updated").

Exogenous Regressors in Forecasts

For forecasting user finances (cash flow, investment growth, etc.), economic indicators often serve as **exogenous regressors** – inputs that influence the forecast outside of the user's direct control. Examples: - Predicting future expenses might include inflation as a regressor (e.g. utility bills might rise with inflation or with energy price index). - Forecasting stock portfolio growth might include interest rates or GDP growth as factors in a model. - Currency fluctuations could affect the value of foreign holdings.

Integration: The forecasting engine (which could be a simple projection or a machine learning model) should be able to pull in indicator data for the relevant period. We define **indicator bindings for forecast models**: - For each forecast model, a mapping of which indicators to use and how. For example, a linear regression model for expenses might have: `expenses_t = α + β1*income_t + β2*inflation_t + ε`. The system would recognize that it needs the inflation rate series as input alongside user's income history. - The binding config could be: `{ indicator: "IPCA_yoy", target: "Forecast.ExpenseModel.inflation" }`, meaning use year-over-year IPCA inflation rate as the variable "inflation" in the ExpenseModel.

At runtime, when generating a forecast: - The system fetches the needed indicator series (say the last N months of YoY inflation for fitting, and future N months of projected inflation for forecasting forward). - It feeds those into the model to produce the forecast.

This approach decouples the model from the data retrieval – the model just knows it has an input called "inflation", and the binding tells which indicator provides that data.

Example: A user's salary growth forecast could be tied to an economic indicator like the national wage index or simply inflation (assuming salaries keep up with inflation). By binding "salaryGrowth = f(inflation)", when the inflation forecast is adjusted, the salary projection automatically shifts.

Scenario Modeling (What-If and ±σ Ranges)

Scenario modeling allows users to explore "what if" situations by tweaking indicators. For instance, *"What if inflation runs 2% higher than expected?"* or *"What if the exchange rate drops to 4 BRL/USD?"* Scenarios can be defined as variations on the baseline indicator values.

Implementing Scenarios: We introduce the concept of **Indicator Sets** in the data model, which can be leveraged for scenarios. An `indicator_set` might represent a coherent collection of indicator values under a certain scenario, with effective date ranges: - The **baseline set** is the default actual data (and perhaps official forecast for future). - A **high inflation scenario** set could be identical to baseline except that future inflation indicator values are, say, +2 standard deviations (σ) higher than baseline each period. - A **low inflation scenario** similarly with -2σ. - A **shock scenario** might alter a specific indicator drastically (e.g. currency devaluation scenario sets USD/BRL +20% from baseline at a future point).

These sets can be stored or generated on the fly. The binding of indicators to features can specify a scenario context. For example, a forecast module can toggle which `indicator_set` to use:

```
forecastEngine.useIndicatorSet("HighInflation");
```

Then all indicator queries within it will fetch from that scenario's values (we can implement this by scoping queries by set, or by dynamically overriding the indicator values post-retrieval).

±σ Range Generation: If the system has an estimate of volatility or error for an indicator forecast, it can automatically generate an upper and lower band (± one standard deviation, or a percentile). These can be shown as range bands in charts. For instance, if baseline 12-month inflation forecast is 5% with σ of 1%, the scenario range is 4% to 6%. We can propagate these through dependent calculations: - A forecast outcome (like projected savings) can be recomputed under the high and low scenarios to produce a range of possible outcomes.

The **binding mechanism** here might link an indicator to multiple scenario variants. Alternatively, scenarios are managed by copying baseline series and modifying values. The `indicator_sets` data contract (see next section) supports an easy switch.

Example Use Case: The user is planning retirement needs. The app shows that under baseline assumptions (inflation ~3% annually, investment return 6%), their savings last 30 years. Under a high-inflation scenario (inflation 6%), maybe the savings only last 25 years, prompting them to save more or invest differently. The subsystem enables this by simply substituting the inflation series and recalculating the longevity of savings.

From an implementation standpoint, scenario modeling largely reuses all the mechanics above – it just picks a different source for indicator values. The heavy lifting is in UI/UX to let the user choose or adjust scenarios, but the backend makes it as simple as selecting a different `source_priority` or dataset for indicators in that scenario.

Data Contracts and Structures

To formalize the above, we define key data structures (TypeScript interfaces and possible storage schema) for indicators, indicator sets, and bindings.

Indicator Data (`indicators`)

This corresponds to the time-series data points of each indicator. In a database, this might be a table `Indicators` or similar. As a TypeScript interface, we showed `IndicatorValue` earlier:

```
interface IndicatorValue {
  code: string;      // e.g. "USD/BRL"
  date: string;      // "YYYY-MM-DD" (for monthlies, perhaps "YYYY-MM-01")
  value: number;
  unit: string;      // "%", "index", "BRL", etc.
  asOf: string;      // timestamp of data version
  source: string;    // source provider name or code
}
```

Primary Key considerations: A combination of `(code, date, asOf)` can uniquely identify a record. To get the latest value, you'd query the max(asOf) for a given code & date. We might also maintain a separate table for the latest values (denormalized for quick access) but that's an optimization.

Indicator Sets (`indicator_sets`)

An `indicator_set` represents a collection of indicators with specified source priorities or overrides, possibly for a certain time span or scenario. It can also be viewed as a **configuration profile** for the indicator subsystem.

Fields could include: - `name` – identifier of the set (e.g. `"Default_Brazil"`, `"HighInflationScenario"`). - `description` – human description if needed (e.g. "Baseline official data for Brazil", "Scenario: Inflation +2%"). - `source_priority_overrides` – this could be a map of indicator code -> custom source list, used if we want a scenario to draw from different data. For example, a scenario might use an alternate forecast for inflation (so for code "IPCA" in that set, source_priority might point to a special provider that yields the higher path). - `effective_dates` – if a set is only applicable for a certain period (e.g. a historical set or a future scenario), we could note a date range.

However, in many cases, `indicator_sets` will be used simply to switch all indicators to a different context (like actual vs user-edited scenario values), so date range might not be needed unless dealing with past methodological changes.

Use case: Suppose the definition of an index changed in 2018 (rebasing, etc). We could have two sets: `CPI_old` for pre-2018 using one source or adjustment, and `CPI_new` for post-2018. The system would choose based on date. But this could also be handled without user-facing sets by internally switching source by date. Still, having the structure allows explicit control.

For simplicity, one can think of `indicator_sets` as akin to environment configuration: - The default set everyone uses normally. - Alternative sets for analysis or testing.

A TypeScript outline:

```
interface IndicatorSetConfig {
  name: string;
  sourceOverrides: Partial<Record<string, string[]>>;
  // e.g. { "IPCA": ["CustomForecast"], "USD/BRL": ["AltFXSource", "BCB"] }
  effectiveFrom?: string;  // start date of applicability
  effectiveTo?: string;
}
```

The system might have a registry of sets. If none is specified, the default (probably with no overrides except using the main catalog defaults) is used.

Indicator Bindings (`bindings`)

Bindings map an indicator (or a transformation of an indicator) to a specific usage in the application. This can be represented in a config file or database that the app reads on startup to know what to wire up. Each binding entry could have:

- `indicator` – the code of the indicator (or possibly an expression like `"IPCA_yoy"` if referring to a derivative, though we can also treat YoY as its own indicator code derived from IPCA).
- `target` – an identifier of the application element or feature it feeds. For example, `"Report.RealToggle"` or `"Budget.Category.Food.inflationRate"` or `"Forecast.ModelX.var2"`.
- `transform` – optional, the type of transformation or role the indicator plays. Could be values like `"deflate"`, `"inflate"`, `"direct"` (use value as is), `"growthRate"`, etc., or a reference to which utility to apply.

The binding can also include **parameters** if needed (like base date for inflation, etc.), but those might be determined at runtime by context (e.g. base date = latest date of report).

Examples of binding entries:

- `{ indicator: "IPCA", target: "Reports.All.realToggleIndex", transform: "deflate" }` – Use IPCA to deflate report values for "real" view.
- `{ indicator: "USD/BRL", target: "PortfolioFX.totalConversion", transform: "direct" }` – Use USD/BRL directly to convert foreign assets to BRL.
- `{ indicator: "SELIC", target: "LoanModel.discountRate", transform: "annualToDaily" }` – Use SELIC rate for loan discounting, applying annual-to-daily conversion internally.
- `{ indicator: "CDI", target: "SavingsSimulation.growth", transform: "compoundDaily" }` – Use CDI daily rate series to compound savings growth.

In implementation, these could be hardcoded connections in code or loaded from a configuration JSON. The benefit of a formal binding config is that it's easier to review and update the relationships without digging through code. It also makes testing easier (we can verify each binding's effect).

Executing bindings: We might have a function that given a `target` and context, retrieves the appropriate indicator and applies transform. However, many bindings will be enacted by specific modules of the app: - The Reports module knows if real toggle is on, it will query the bound inflation index and adjust values. - The Budget module, when projecting forward, knows to look for an inflation uplift binding for each category. - The Forecast engine reads its config of which regressors (indicators) to include.

So the binding entries serve as documentation and configuration for developers and as a reference for automated tests (e.g. ensuring every required binding has data).

Reliability & Operations

To maintain trust in the system's outputs, we need processes for monitoring data freshness, handling updates, and tracing data usage.

Alerting on Missing or Stale Data: The subsystem should include checks that run periodically to ensure data is up-to-date: - For each indicator, compare the last date we have vs today (or vs expected last date). If a daily series hasn't updated in more than 1 day (and today isn't a weekend/holiday), raise an alert. For monthly series, if it's past the usual release day of the month and no new data, flag it. - The alerting mechanism could be as simple as console logs for a developer, or integration with a monitoring service to send an email/Slack message if data is stale beyond a threshold. This is important for critical indicators like FX and interest rates. - Also alert if a provider fails to fetch data for X attempts in a row, indicating the API might be down or keys expired, etc.

Automated Re-fetch on Schedule: We can set up cron-like jobs: - Daily early morning (or after typical publish times) to fetch previous day's closing values for daily series (so data is ready when users open the app). - Monthly on known release dates for indices (e.g. around 10th for IPCA, etc.). - This proactive fetching populates the cache so users don't experience lag when viewing data.

Recompute Forecasts on Data Revision: If an indicator value that has already been used in forecasts or reports gets revised, we have two choices: - If we aim for strict reproducibility, we might not *automatically* update past forecasts (because those were made with data available at the time). Instead, we'd notify that new data is available and allow user to recompute forecasts. - However, for forward-looking projections, we likely do want the latest data. So a reasonable approach: when a revision or new data point comes in: - Mark any saved forecasts or analyses that depended on that indicator as "stale". This could be done via a simple version number or timestamp. For example, each forecast result could store `lastDataAsOf`. If new data has `asOf` greater, then forecast is stale. - If the app is largely doing on-the-fly calculations, it can just pull the latest data next time it runs. But if we store forecast outputs (say a cached simulation result), we either invalidate or recompute them. - Recomputing can be automated for crucial things. For instance, if the user has an active long-term projection, the app might auto-recalculate it in background when data updates, so that when they open it, it's up to date.

Audit Trail of Usage: It's valuable to log or record what data was used in user outputs, especially for financial projections. This can be as detailed as: - Storing the `asOf` date (or even the exact version IDs) of indicators within each forecast or report snapshot. E.g., if we save a PDF report for the user, embed "Data as of 2025-09-30" in it, or even list key assumptions. - Keeping a changelog: e.g., "Oct 15, 2025: Revised IPCA Sep/2025 from 0.34% to 0.36%, updated forecasts accordingly." - If disputes or questions arise ("why did my retirement projection change?"), this audit trail helps explain that the underlying inflation or interest assumption changed due to new data.

From an implementation standpoint, maintaining an audit trail could be as simple as writing entries to a log file or database table whenever indicator values are fetched/updated and whenever forecasts are generated. For example:

```
2025-10-15 10:00: IPCA Sep/2025 value revised from 8120.45 to 8120.60 (asOf
2025-10-15). Marking Forecast#123 as stale.
2025-10-15 10:05: Forecast#123 recomputed with new IPCA data (asOf 2025-10-15).
Outcome changed from X to Y.
```

These logs could be exposed in a debug section of the app or just kept internally.

Handling Downtime and Backups: In ops, consider: - If a provider is known to have maintenance downtime, perhaps schedule cache refreshes accordingly or extend TTL over that period. - Keep local backups of critical series (the snapshot provider idea) so that even if the entire upstream is down, users see last known info rather than an error. - Provide a manual override mechanism: e.g., if an indicator will be discontinued or changed, ops can manually feed data or switch sources quickly via config (taking advantage of our provider-agnostic design).

Quality Assurance and Testing

Ensuring the correctness of the economic indicator subsystem is paramount, given that financial decisions may rely on it. We outline a QA checklist and a testing strategy (including golden files and snapshot tests):

QA Checklist:

1. **Data Accuracy Verification:**
2. For each indicator, verify that the values fetched match the official source for a set of test dates. For example, cross-check that the last 3 months of IPCA values in the system equal the numbers on the IBGE/BCB release [10] . Automate this check if possible (e.g., via secondary source or known constants).
3. Ensure units and scaling are correct (no off-by-100 errors for percentages, etc.).
4. **Schema and Metadata Consistency:**
5. The catalog entries should correctly describe the indicators. Write tests that for each indicator code, any sample data fetched aligns with its `frequency` (e.g., no duplicate dates for monthly series, daily series have only weekdays if expected, etc.), and `unit` (e.g., if unit is "%", values are within a plausible range 0–100 or a small number if using 0.05 for 5% convention).
6. If `seasonal_adjusted=false` for an indicator, verify that if an alternate seasonally adjusted series exists, it's a separate code to avoid confusion.
7. **Provider Failover Logic:**
8. Simulate a primary provider failure and ensure the system correctly falls back to secondary. This can be done by mocking the provider responses. The expected outcome: data still returns via fallback, and a log or flag indicates primary failed.
9. Simulate all providers failing: the function should throw an error or return cached data with a warning. Confirm that behavior.
10. **Caching Behavior:**
11. Test that within a short period, repeated requests do not trigger new fetches (e.g. call getIndicatorData twice and ensure the second time it hit cache, perhaps by injecting a counter in provider).
12. Test that after TTL expires or after new data date, the next request does fetch new data.

13. If applicable, test offline mode: disconnect network or force providers to throw, and verify that the system serves data from the snapshot cache without crashing.

14. **Normalization Functions:**

15. Write unit tests for inflation adjustment: e.g., known scenario: if inflation index doubled from 50 to 100, an amount of 200 should adjust to 400. Use small synthetic index series to validate `adjustForInflation` correctness.
16. Test interest conversions: using a known annual rate (say 0%), expect daily = 0%; use a rate like 100% annual, verify daily approx 0.002726 (since (1+1)^(1/252)-1). We can use the formula outside to assert our function matches.
17. Test accumulate and discount functions with simple inputs (e.g. 10 days of 0.1% daily should roughly equal 1% total).

18. FX conversion: test converting A->B->A returns original (within rounding), using a fixed rate. E.g., if 1 USD = 5 BRL, 5 BRL to USD should give 1, and back to BRL gives 5.

19. **Binding Integration:**

20. Simulate the real vs nominal toggle: prepare a dummy report dataset and apply toggle both ways, verify that when toggled to real, each data point = nominal * (baseIdx/idx) using a test index series. Basically confirm the deflate logic is correctly wired.
21. Budget inflation uplift: for a test budget item, mark it inflation-adjusted, simulate an inflation forecast of say 10%, and ensure next year's budget value computed = current * 1.1. If user toggles it off, it stays = current (or uses user's own value).

22. Forecast regressor: plug in a dummy model (e.g. a linear regression with known coefficients) and feed it through the binding mechanism with an indicator series, check that the output uses actual indicator data. This might be done by injecting a fake forecast function that records its inputs.

23. **Scenario Switching:**

24. Create a baseline and an alternate indicator set for a small test series (e.g. baseline inflation = 3%, scenario = 6%). Run a calculation (like a 10-year compounded price increase) under both and verify the scenario's result is higher as expected.

25. If the UI allows easy scenario toggle, simulate that and ensure the data feed actually switched (which could be confirmed by checking which provider or which values returned).

26. **Performance under load:** (if needed)

27. Ensure that fetching all indicators (e.g. on app startup if it pulls many series) is reasonably fast with caching. This might not be unit test but a QA step to profile.

Golden File and Snapshot Testing:

A **golden file** is a reference output saved from a known good run, used to detect changes. We can use this technique for our subsystem: - After implementing, fetch a broad set of data (all indicators for a certain date

range) and save it as JSON (the golden dataset). On subsequent test runs (or before a release), fetch again and compare to the golden. Differences might indicate a breaking change or simply updated data. We need to distinguish expected differences (e.g. new data or legitimate revisions) from unexpected (bug in parsing or provider change). - For transformations, have golden outputs as well. For instance, feed known inputs to the inflation adjust function and store the outputs. If an internal change alters these, flag it.

Because economic data does change over time, we likely update golden files periodically. One approach is to fix a cut-off date for golden data. For example, we maintain a golden file for all values up to Dec 31, 2024. That data won't change (assuming we captured final revisions). Our tests compare the subsystem's data for <=2024 with the golden. They should match exactly. This catches any parsing or storage bugs (like mis-ordering or truncation) for historical data. For post-2024 data, we can't have a stable golden because it's updating; instead, we focus on dynamic checks (like monotonicity or range, as above).

Snapshot Tests for UI Integration: - We can also take snapshots of rendered output in features. For example, generate a sample report in nominal and real terms and save the values. If a future code change or data change causes a large deviation in real vs nominal calculations, the snapshot test would show it. The team can then confirm if it's expected (e.g. inflation data updated) or a bug. - Similarly, a forecast result with given assumptions can be snapshotted. If changing the indicator set or any logic changes the forecast unexpectedly, test fails.

Manual Testing: - Before release, manually switch the app between nominal/real, scenarios, etc., to see that numbers make intuitive sense (e.g. in real mode things should generally be lower in the past due to inflation, etc.). - Manually simulate a provider outage (perhaps by disabling internet and clearing cache) to see the app still functions in offline mode with cached data.

By following this comprehensive QA approach, we ensure the economic indicator subsystem is accurate, reliable, and performs as expected. The combination of automated tests (unit, integration, snapshot) and monitoring in production (alerts on stale data) will catch issues early and give confidence in the subsystem's correctness.

Conclusion

This implementation guide has detailed the design of a provider-agnostic economic indicator subsystem for a TypeScript/React/Next.js application. We defined how indicators are cataloged with metadata, how multiple providers are abstracted and managed with caching and validation, and how critical transformations are implemented (inflation adjustment, interest compounding, FX conversion). We also described how these indicators bind into application features like reports, budgets, and forecasts to enhance them with real economic context. Lastly, we covered data contracts and rigorous strategies for reliability, including versioning for reproducibility [2] and thorough testing practices.

By adhering to these designs, the personal finance app will be equipped with a flexible and robust indicators module. It will be easy to extend (adding new countries or sources), resilient to data issues (with fallbacks and alerts), and will greatly enrich user experience by grounding personal finances in the real economic environment. As the app grows globally, this subsystem can scale by simply expanding the catalog and plugging in new providers, following the patterns established here. Each component – from data fetch to transformation to binding – is modular, testable, and maintainable, ensuring that both developers and users can trust the economic insights provided.

Sources:

- Brazil Central Bank data (examples of indicators provided via API) [1]
- Steven V. Miller – *Adjusting Data for Inflation (Real vs Nominal)* [4] [11]
- Reddit /investimentos – *CDI daily rate calculation (annual to daily conversion)* [7]
- St. Louis Fed ALFRED – *Rationale for vintage data storage (revisions and reproducibility)* [2]

---

[1] Data series from Brazilian Central Bank | by Antelo | Medium
https://antelo.medium.com/get-data-series-from-bcb-banco-central-do-brasil-30dc1eb63fea

[2] [8] [10] Help | ALFRED | St. Louis Fed
https://alfred.stlouisfed.org/help

[3] [PDF] Applying the daily inflation to forecast the Broad Consumer Price ...
https://stats.unece.org/ottawagroup/download/f420.pdf

[4] [5] [6] [9] [11] How to Adjust for Economic Indicators for Inflation (and Index Them) | Steven V. Miller
http://svmiller.com/blog/2023/01/index-economic-data-adjust-inflation/

[7] A question about the variation of the DI and CDB rates. : r/investimentos
https://www.reddit.com/r/investimentos/comments/1jfcq5u/uma_d%C3%BAvida_com_a_varia%C3%A7%C3%A3o_da_taxa_di_e_cdb/?tl=en