



INTRODUCTION

Mock Magic is a zero-code mock data generator for Unity. It produces realistic fake data - names, emails, addresses, time series, streaming feeds, and 50+ other data types - directly from an Editor window. No scripting required.

Open **Tools** → **Mock Magic** → **Generator**, pick a data type, and copy the results. That's it.

Mock Magic covers three domains of test data:

- **Simple data** - individual values like names, emails, phone numbers, GUIDs, lorem ipsum, and more (55 types in total).
- **Series data** - patterned datasets for charts, graphs, and tabular testing (15 mathematical patterns with configurable noise, labels, and multi-series support).
- **Streaming data** - continuous real-time data emission with configurable rhythms, usable in both Edit Mode and Play Mode.

All generated data can be exported to **Text**, **CSV**, **JSON**, or **C# code** and copied to the clipboard with a single click. For developers who need runtime data generation, Mock Magic also offers a clean scripting API.



CONTENTS

Introduction	1
Key Features	3
Swift Charts Parity	3
<hr/>	
Getting Started	4
Simple Data	5
Series Data.....	7
Streaming Data	10
Exporting Data	12
Using MockMagic with Chart Guru	14
Scripting API	16
FAQ.....	25
I would like to add an additional type	25
<hr/>	
Support	26



KEY FEATURES

Feature	Description
55 data types	Personal info, contact details, locations, business data, dates, text, numbers, technical identifiers, and more
15 series patterns	Random, Linear, Exponential, Logarithmic, Sine, Cosine, TrendingUp, TrendingDown, Seasonal, Stepped, Sparse, RandomWalk, BellCurve, Sawtooth
17 label presets	Months, Weekdays, Quarters, Years, Products, Regions, Teams, Departments, Countries, Cities, Colors, Letters, and more
4 streaming rhythms	Constant, Random, Burst, Drip - with full timing control
4 export formats	Text, CSV, JSON, C# code - one-click clipboard copy
Visual Editor Window	Four-tab interface: Simple, Series, Streaming, Export - no code needed
Edit Mode & Play Mode	Streaming works in both modes - test without entering Play Mode
Seedable determinism	MockRandom facade with Mock.Seed, Mock.UseDateReference, and a Random / Stable toolbar in the generator window
{{token}} templating	Mock.Parse("{{firstName}} <{{email}}>") plus Mock.RegisterToken / RegisterGenerator for custom types
Object builders	MockPerson (consistent person record) and MockOf<T> for fluent, AOT-safe object population
Chart Guru integration	Chart Guru includes a built-in <i>MockChartDataSource</i> component powered by Mock Magic

SWIFT CHARTS PARITY

Mock Magic's data structures are designed to map directly to charting frameworks like Apple's Swift Charts and Chart Guru. A *MockDataPoint* with *Label*, *Value*, and *SeriesName* translates one-to-one to a *BarMark*, *LineMark*, or *AreaMark*. The 14 series patterns cover every standard chart scenario - from simple bar charts (Random, Stepped) to financial dashboards (RandomWalk) to scientific plots (BellCurve, Sine). Generate the data in Mock Magic, export it, and use it directly.



GETTING STARTED

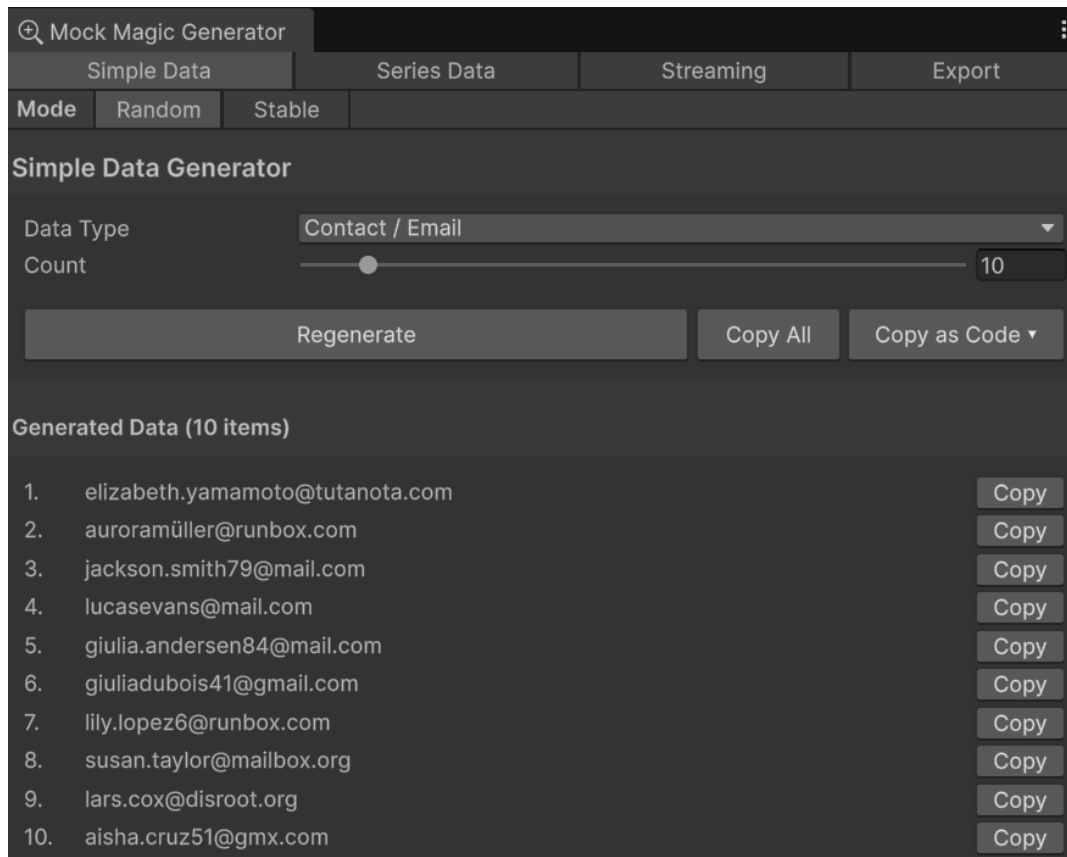
Installation

1. Install MockMagic from the Package Manager.
2. Dependencies are resolved automatically.
3. Verify installation: open **Tools** → **Mock Magic** → **Generator** from the Unity menu bar.

Your First Mock Data - 30 Seconds

1. Open **Tools** → **Mock Magic** → **Generator**.
2. The **Simple Data** tab is selected by default.
3. Click the **data type dropdown** - a categorized menu appears with groups like Personal, Contact, Location, and more.
4. Select **Contact** → **Email**.
5. Click **Copy All** to copy everything to your clipboard.

That's it. You now have 10 fake email addresses ready to paste wherever you need them.



The Four Tabs

The Mock Magic Generator window has four tabs, each serving a distinct purpose:

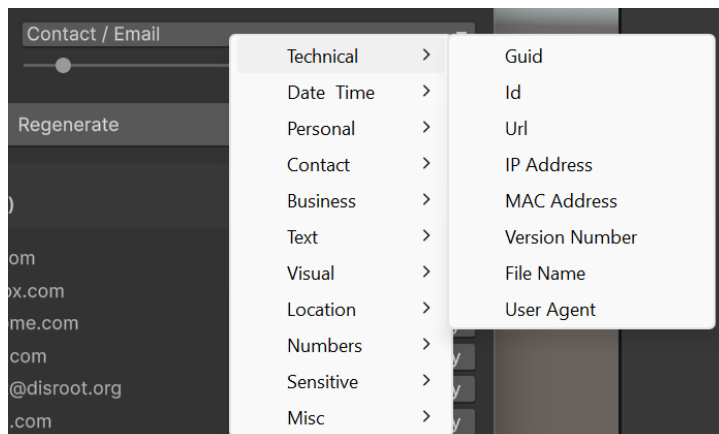
Tab	Purpose
Simple Data	Generate individual fake values - names, emails, dates, colors, UUIDs, and 45+ more types
Series Data	Generate patterned datasets for charts and graphs - trending, seasonal, bell curves, etc.
Streaming	Emit mock data continuously in real-time - test live dashboards and feeds
Export	Format and copy generated data as Text, CSV, JSON, or C# code

SIMPLE DATA

The Simple Data tab generates individual fake values or batches of values from 51 built-in data types. Every value is generated from realistic embedded datasets - names are multicultural, emails have plausible domains, addresses follow real formats.

Using the Simple Data Tab

1. **Data type dropdown** - Click to open a categorized menu. Types are grouped into 11 categories for easy discovery.
2. **Count slider** - Set how many values to generate (1–100).
3. **Regenerate** - Click to generate a new batch. Data also regenerates automatically when you change the type or count.
4. **Copy All** - Copies all generated values to the clipboard, separated by newlines.
5. **Preview list** - A scrollable list showing all generated values, numbered. Each item has its own **Copy** button for copying a single value.



Note: All generated data is fake. Credit card numbers pass the *Luhn* check but are not real. ISBN check digits are valid. Passwords use *Fisher-Yates* shuffling with guaranteed uppercase, lowercase, digit, and special character.

Tips

- **Combine types for realistic records.** Generate 10 names, 10 emails, and 10 cities separately, then pair them together for a mock user database.
- **Use Copy All for bulk pasting.** The clipboard output is newline-separated - paste directly into spreadsheets, text files, or code editors.
- **Adjust the count slider** for anything from a single value (quick placeholder) to 100 items (stress testing).

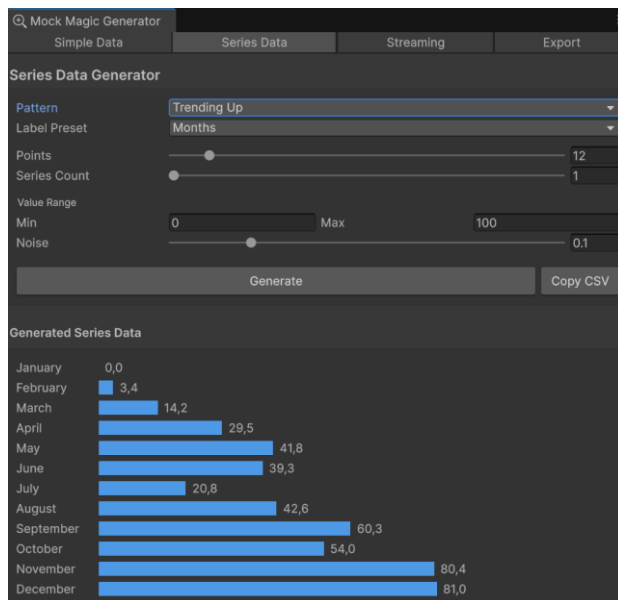


SERIES DATA

The Series Data tab generates patterned numerical datasets - the kind of data you feed into charts, graphs, dashboards, and data visualizations. Each dataset consists of labeled data points with values that follow a mathematical pattern.

Using the Series Data Tab

1. **Pattern** - Select from 14 mathematical patterns (see [Pattern Reference](#) below).
2. **Label Preset** - Choose how data points are labeled on the X-axis: months, weekdays, quarters, numeric indices, and more.
3. **Points** - Set how many data points per series (3–100).
4. **Series Count** - Generate multiple series at once (1–10). Useful for grouped/stacked charts or comparisons.
5. **Min / Max** - Define the value range for generated data.
6. **Noise** - Add randomness to the pattern (0 = perfectly clean, 0.5 = very noisy). Noise makes data look more realistic.
7. **Generate** - Click to produce the dataset. A visual bar preview appears below.
8. **Copy CSV** - Copies the generated data in CSV format to the clipboard.



Visual Preview

After generating, the Series Data tab displays an inline horizontal bar chart. Each data point shows its label, a colored bar proportional to the value, and the numeric value. For multi-series data, series are separated by name headers.



The Data Point Structure

Every generated series data point is a MockDataPoint with five fields:

Field	Description
Label	The X-axis category label (e.g., "Jan", "Q1", "North")
Value	The primary Y-axis value
SecondaryValue	An optional secondary value, useful for range/band charts
SeriesIndex	Which series this point belongs to (0-based)
SeriesName	The name of the series (e.g., "Revenue", "Series 1")

Scripting: Access these fields directly - point.Label, point.Value, point.SeriesName, etc.

Pattern Reference

Pattern	Description
Random	Uniformly random values between min and max. Good for bar charts and scatter data.
Linear	Steady progression from min to max. A clean diagonal line.
Exponential	Slow start that accelerates sharply upward. Growth curves and compound metrics.
Logarithmic	Fast initial rise that flattens out. Diminishing returns, learning curves.
Sine	Smooth wave oscillating around the midpoint. Cyclic patterns, audio-like data.
Cosine	Same as Sine but phase-shifted by 90°. Starts at the peak.
TrendingUp	General upward movement with random walk noise. Revenue growth, user acquisition.
TrendingDown	General downward movement with random walk noise. Decline metrics, churn.
Seasonal	Sine wave combined with a linear upward trend. Monthly sales with yearly seasonality.
Stepped	Staircase pattern with flat plateaus. Pricing tiers, plan upgrades.
Sparse	70% zero values, 30% random non-zero spikes. Event data, error logs.



Pattern	Description
RandomWalk	Cumulative random steps from the midpoint. Stock prices, financial data.
BellCurve	Normal distribution shape centered at the midpoint. Survey results, test scores.
Sawtooth	Repeating ramp-up pattern. Periodic resets, charge/discharge cycles.
GaussianNoise	Random noise drawn from a Gaussian (normal) distribution centred on the midpoint. Useful for sensor jitter and measurement-error simulations.

Multi-Series Data

Set **Series Count** to more than 1 to generate multiple series sharing the same labels. Each series follows the selected pattern independently but uses the same label axis. This is ideal for:

- **Grouped bar charts** - comparing categories across series
- **Multi-line charts** - overlaying trends
- **Stacked area charts** - showing composition over time

Series are automatically named "Series 1", "Series 2", etc. unless custom names are provided.

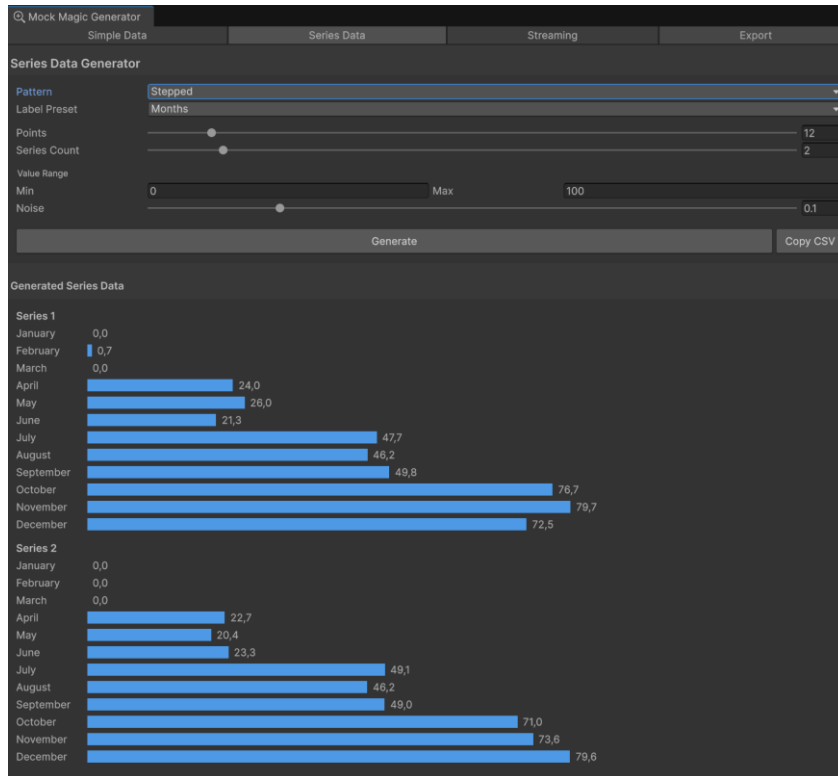
Understanding Noise

The **Noise** slider (0–0.5) adds controlled randomness to any pattern:

- **0** - Perfectly clean mathematical pattern. Useful for demonstrations or when you need exact shapes.
- **0.05–0.15** - Subtle variation. Data looks realistic without losing the pattern shape. This is the sweet spot for most chart testing.
- **0.3–0.5** - Heavy noise. The underlying pattern is still visible but data looks rough and organic.

All values are clamped to the min/max range after noise is applied, so noise never produces out-of-bounds values.





STREAMING DATA

The Streaming tab provides continuous real-time mock data emission. Instead of generating a static dataset, it emits new data points at configurable intervals - perfect for testing live dashboards, real-time charts, monitoring UIs, or any system that consumes a data feed.

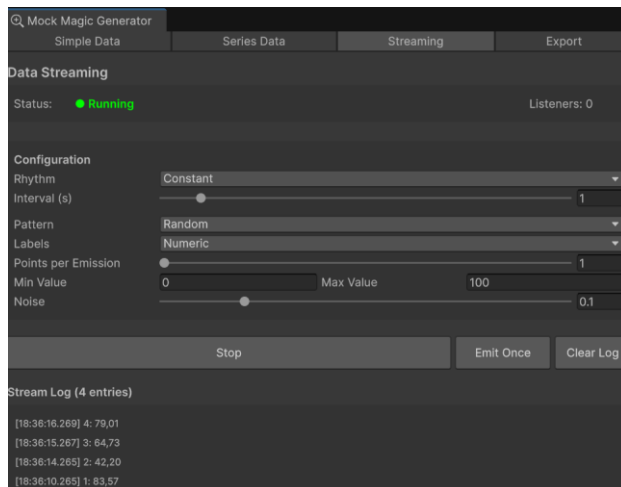
Streaming works in both **Edit Mode** and **Play Mode**. You don't need to enter Play Mode to test a live data feed.

Using the Streaming Tab

1. **Status indicator** - Shows "● Running" (green) when streaming or "○ Stopped" (gray) when idle. Also displays the current listener count.
2. **Rhythm** - Select how data is emitted over time (see Rhythm Reference below). The controls below update dynamically based on the selected rhythm.
3. **Pattern** - The series pattern used for generated values (same 14 patterns as the Series tab).
4. **Labels** - The label preset for emitted data points.
5. **Points per Emission** - How many data points are generated per emission (1–10).
6. **Min / Max Value** - Value range for generated data.



7. **Noise** - Randomness factor (0–0.5).
8. **Start / Stop** - Toggle button to begin or end streaming.
9. **Emit Once** - Fire a single emission manually without starting continuous streaming.
10. **Clear Log** - Clear the stream log.
11. **Stream Log** - A scrollable list of timestamped entries showing each emitted data point. Newest entries appear at the top. Maximum 50 entries.



Rhythm Reference

Rhythm	Behavior
--------	----------

Constant	Emits at a fixed interval. Predictable, metronome-like.
----------	---

Random	Emits at random intervals within a range. More organic feel.
--------	--

Burst	Emits a rapid burst of data, then pauses. Simulates batch updates.
-------	--

Drip	Emits single values one at a time. Slow, steady feed.
------	---

Label Continuity

Labels continue sequentially across emissions. If the first emission produces "Jan, Feb, Mar" and the next emission produces 3 more points, they will be labeled "Apr, May, Jun" - not "Jan, Feb, Mar" again. This makes streaming data behave realistically over time.

Listener Model

External systems receive stream data by registering as listeners. The **Status indicator** in the Streaming tab shows the current listener count. The MockMagic Editor window itself is always registered as a listener (that's how the Stream Log works). Other listeners - such as charts, scripts, or monitoring systems - can register via the scripting API (see Section 8).



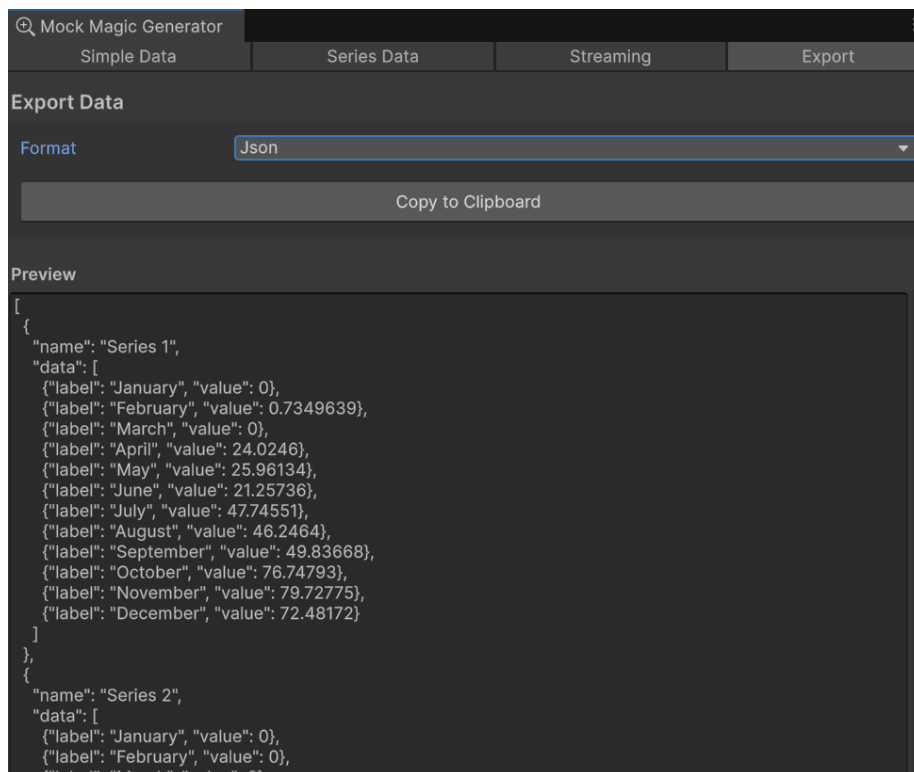
EXPORTING DATA

The Export tab formats your generated data for use outside MockMagic. Whether you need to paste test data into a spreadsheet, feed it to an API, or embed it directly in C# code, the Export tab handles the conversion and copies the result to your clipboard.

Using the Export Tab

1. **Format dropdown** - Select the output format: Text, CSV, JSON, or C# Code.
2. **Variable Name** - (*Shown only for C# Code format.*) Set the variable name used in the generated C# array declarations.
3. **Copy to Clipboard** - Click to copy the formatted output.
4. **Preview** - A scrollable text area showing the full formatted output. Updates automatically when you change the format or variable name.

If no data has been generated yet, the tab shows a help message: "Generate some data first."



Data Priority

The Export tab automatically uses the most recent data you generated, in this priority order:

1. **Multi-series data** (from the Series tab with Series Count > 1)
2. **Single series data** (from the Series tab with Series Count = 1)
3. **Simple data** (from the Simple Data tab)



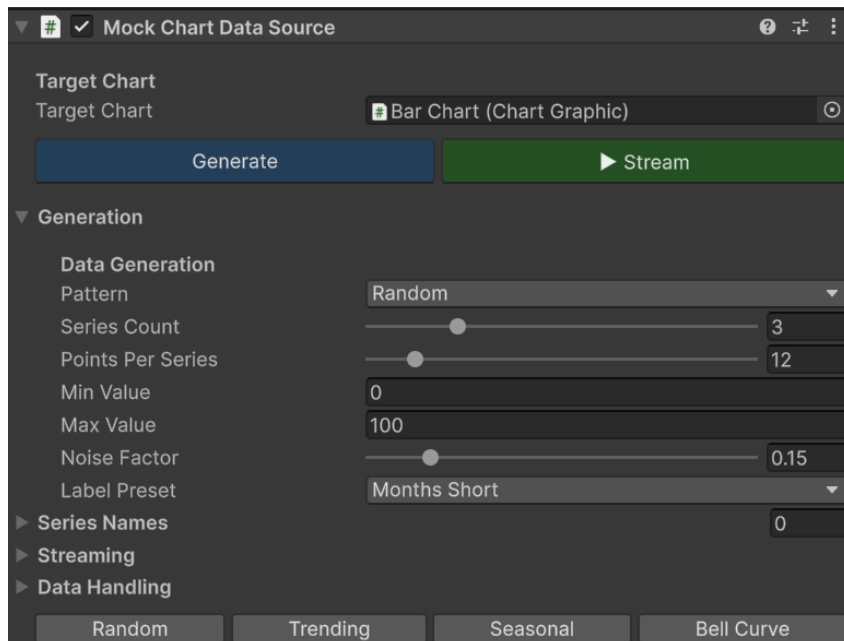
USING MOCKMAGIC WITH CHART GURU

Chart Guru ships with a built-in component called *MockChartDataSource* that is powered entirely by MockMagic. You do not need to open the MockMagic Generator window, export data, or write any code - just add the component to your chart and configure it in the Inspector.

Adding MockChartDataSource

There are two ways to add it:

1. **From the ChartGraphic Inspector** - When a ChartGraphic component has no data source attached, the Inspector shows an **"Add Mock Data Source"** button. Click it to add MockChartDataSource automatically.
2. **From the Add Component menu** - Select **Add Component** → **ChartGuru** → **Mock Chart Data Source**.



Works in Edit Mode

MockChartDataSource works in both Edit Mode and Play Mode. You can:

- Click **Generate** to preview chart data without entering Play Mode
- Start **streaming** to see live chart updates in the Scene view
- Swap patterns and label presets with instant visual feedback

This makes it an excellent rapid prototyping tool - configure your chart's look and feel with realistic data before writing a single line of code.

Under the Hood

MockChartDataSource uses MockMagic's *MockSeries* class internally:

- **One-shot generation** calls *MockSeries.Generate()* (single series) or *MockSeries.GenerateMultiSeries()* (multi-series)
- **Streaming** calls *MockSeries.GenerateLabels()* and *MockSeries.GenerateValues()* to produce incremental data points
- Labels continue sequentially across emissions (same continuity behavior as MockMagic's Streaming tab)

Alternative Workflow: Export and Paste

If you prefer to use Chart Guru's *ManualChartDataSource* instead, you can:

1. Open the MockMagic Generator window (**Tools** → **Mock Magic** → **Generator**)
2. Generate series data in the **Series Data** tab
3. Switch to the **Export** tab and select **CSV** or **JSON**
4. **Copy to Clipboard**
5. Paste into your data source, configuration file, or script

This workflow gives you more control over the exact data, but requires an extra step compared to the integrated *MockChartDataSource* component.



SCRIPTING API

For developers who need mock data at runtime - automated testing, procedural content, live dashboards, or CI pipelines - MockMagic provides a clean static API. Every feature available in the Editor window is also available from code.

Assembly reference: Add a reference to the MockMagic assembly definition in your *.asmdef* file, or place your scripts in a folder without an assembly definition.

Namespace: using MockMagic;

Simple Data from Code

The Mock static class provides one method per data type, plus batch variants.

Single Values

using MockMagic;

```
string name = Mock.Name();           // "Sarah Johnson"
string email = Mock.Email();         // "marcus.patel@outlook.com"
string phone = Mock.Phone();         // "+1 (425) 738-2914"
int age = Mock.Age();                // 34
string address = Mock.FullAddress(); // "1847 Oak Ave, Seattle, WA 98101"
string guid = Mock.Guid();           // "a3f8b2c1-d4e5-4f6a-8b9c-0d1e2f3a4b5c"
float pct = Mock.Percentage();       // 87.45
bool flag = Mock.Boolean();          // true
string lorem = Mock.LoremIpsum(20); // 20 words of lorem ipsum
UnityEngine.Color color = Mock.Color();// Random RGB color
```

Batch Values

Every single-value method has a batch counterpart that returns an array:

```
string[] names = Mock.Names(50);
string[] emails = Mock.Emails(100);
int[] ages = Mock.Ages(25);
string[] cities = Mock.Cities(10);
```



Generic Dispatch

Use `Mock.Generate()` when the data type is determined at runtime:

```
// Single value
```

```
string value = Mock.Generate(SimpleDataType.Email);
```

```
// Batch
```

```
string[] values = Mock.Generate(SimpleDataType.Email, 50);
```

Random Enum Values

Generate random values of any enum type:

```
MyEnum randomValue = Mock.Enum<MyEnum>();
```

```
MyEnum[] randomValues = Mock.Enums<MyEnum>(10);
```

ID Counter

`Mock.Id()` returns auto-incrementing integers. Reset with:

```
Mock.ResetIdCounter(0); // Next Id() call returns 0
```

Determinism & Seeding

`Mock.Id()` returns auto-incrementing integers. Reset with:

```
Mock.ResetIdCounter(0); // Next Id() call returns 0
```

Call `Mock.Seed(int)` once at the top of a deterministic run; pair it with `Mock.UseDateReference(...)` so date-, time- and timestamp-based helpers stay reproducible across days.

```
Mock.Seed(12345); // seeds MockRandom + UnityEngine.Random
Mock.UseDateReference(new DateTime(2026, 1, 1)); // freeze "now" for date helpers
string a = Mock.Email(); // reproducible
string b = Mock.Date(); // reproducible
Mock.ResetSeed(); // back to wall-clock randomness
```

Format & Hash Helpers

Shape any string with a tiny pattern language: '#' = digit, '?' = lowercase letter, '*' = alphanumeric, '\' = escape the next character. Phone, SSN, ZIP and MAC helpers all use it internally.

```
string sku = Mock.Format("AB-####-???"); // "AB-2741-xqp"
string sha = Mock.Hash(); // 40-char hex (SHA-1 length)
string code = Mock.AlphaNumeric(8); // "k4j9pq7m"
string tag = Mock.Letters(5, upperCase: true);
```



Template Tokens & Custom Generators

Every *SimpleDataType* is a built-in token. Unknown tokens are left in place so they're easy to spot during debugging. Register your own tokens - or full typed generators - to extend the system without forking the package.

```
string row = Mock.Parse("{{firstName}} <{{email}}> id={{unique}}");
```

```
Mock.RegisterToken("orderRef", () => "ORD-" + Mock.Format("#####"));
Mock.RegisterGenerator<int>("score", () => MockRandom.Range(1, 101));
int s = Mock.Get<int>("score");
string r = Mock.Parse("Reference: {{orderRef}}");
```

Probabilistic Nulls

Drop-in nullability for stress-testing consumers:

```
string maybeAddress = Mock.FullAddress().OrNull(0.3f); // reference types
int? maybeAge = Mock.Age().OrNullable(0.2f); // value types -> Nullable<T>
float maybeScore = Mock.Float().OrDefault(0.1f); // any T -> default(T)
```

Object Builders

MockPerson.New() returns an internally consistent record - email and username are derived from the drawn name, the date of birth agrees with Age, and the address parts cohere into a single line.

```
MockPerson p = MockPerson.New();
MockPerson[] roster = MockPerson.New(50);
```

MockOf<T> is a delegate-based fluent builder for any POCO. No reflection, no expression trees - IL2CPP/AOT safe.

```
User[] users = MockOf<User>
    .New(() => new User())
    .Set((u, i) => u.Id = i + 1)
    .Set(u => u.Name = Mock.Name())
    .Set(u => u.Email = Mock.Email())
    .UseSeed(42)
    .Generate(100);
```

Image & Themed Text Helpers



*Mock.ImageUrl(width, height) - random picsum.photos URL.

*Mock.PlaceholderImageUrl(width, height, bg, fg, text) - placeholder.co URL with optional colours and label.

*Mock.SvgDataUri(width, height, fill, label) - self-contained, offline SVG data URI.

*Mock.HackerPhrase() - tech-flavoured activity-log line, e.g. "Try parsing the SSL sensor, maybe it will reboot the digital bus!".

*Mock.Review(minWords, maxWords) - short product-style review string for e-commerce mocks.

Series Data from Code

The MockSeries static class generates patterned datasets.

Full-Featured Generation

using MockMagic;

```
MockDataPoint[] data = MockSeries.Generate(  
    pattern: SeriesPattern.TrendingUp,  
    count: 12,  
    min: 0f,  
    max: 100f,  
    noise: 0.15f,  
    labelPreset: LabelPreset.MonthsShort  
);
```

// Each point has: data[i].Label, data[i].Value, data[i].SeriesName, etc.

Multi-Series Generation

```
MockDataPoint[][] multiData = MockSeries.GenerateMultiSeries(  
    pattern: SeriesPattern.Seasonal,  
    seriesCount: 3,  
    pointCount: 12,  
    min: 0f,  
    max: 100f,  
    noise: 0.1f,  
    labelPreset: LabelPreset.MonthsShort,  
    seriesNames: new[] { "Revenue", "Costs", "Profit" }  
);
```

// multiData[0] = Revenue series, multiData[1] = Costs series, etc.

Convenience Methods



Short-form methods with sensible defaults:

```
MockDataPoint[] data = MockSeries.Linear();    // 12 points, 0–100, low noise
MockDataPoint[] data = MockSeries.Sine();      // Sine wave
MockDataPoint[] data = MockSeries.TrendingUp(); // Upward trend
MockDataPoint[] data = MockSeries.TrendingDown(); // Downward trend
MockDataPoint[] data = MockSeries.Seasonal();  // Seasonal pattern
MockDataPoint[] data = MockSeries.BellCurve(); // Normal distribution
MockDataPoint[] data = MockSeries.Random();    // Random values
MockDataPoint[] data = MockSeries.Stepped();   // Staircase
MockDataPoint[] data = MockSeries.Sparse();    // 70% zeros
MockDataPoint[] data = MockSeries.RandomWalk(); // Financial-style random walk
MockDataPoint[] data = MockSeries.Exponential(); // Exponential growth
```

Raw Values Only

If you only need the float array without labels:

```
float[] values = MockSeries.GenerateValues(
    SeriesPattern.Sine, count: 50, min: -1f, max: 1f, noise: 0f
);
```

Custom Labels

```
string[] labels = MockSeries.GenerateLabels(
    LabelPreset.MonthsShort, count: 12, startIndex: 0
);
// ["Jan", "Feb", "Mar", ..., "Dec"]
```

Streaming from Code

For continuous real-time data, use the `MockStreaming` static convenience class or the `MockStream MonoBehaviour` singleton directly.



Using MockStreaming (Recommended)

```
using MockMagic;  
using UnityEngine;
```

```
public class LiveDashboard : MonoBehaviour  
{  
    private void OnEnable()  
    {  
        // Configure the stream  
        MockStreaming.Configure(  
            pattern: SeriesPattern.RandomWalk,  
            min: 0f,  
            max: 100f,  
            noise: 0.1f,  
            pointsPerEmission: 1,  
            labelPreset: LabelPreset.MonthsShort  
        );  
  
        // Register a listener  
        MockStreaming.Register(OnDataReceived);  
  
        // Start streaming  
        MockStreaming.Start(interval: 1.0f, rhythm: StreamRhythm.Constant);  
    }  
  
    private void OnDisable()  
    {  
        MockStreaming.Unregister(OnDataReceived);  
        MockStreaming.Stop();  
    }  
  
    private void OnDataReceived(MockDataPoint[] points)  
    {  
        foreach (MockDataPoint point in points)  
        {  
            Debug.Log($"{point.Label}: {point.Value}");  
        }  
    }  
}
```



Using MockStream Directly

MockStream is a singleton MonoBehaviour with full property access:

```
MockStream stream = MockStream.Instance;  
stream.Pattern = SeriesPattern.Sine;  
stream.Rhythm = StreamRhythm.Burst;  
stream.Interval = 0.5f;  
stream.BurstCount = 5;  
stream.BurstDelay = 3f;  
stream.MinValue = 0f;  
stream.MaxValue = 100f;  
stream.Noise = 0.1f;  
stream.PointsPerEmission = 3;  
stream.LabelPreset = LabelPreset.WeekdaysShort;
```

```
stream.OnDataEmitted += OnDataReceived;  
stream.OnStreamStarted += () => Debug.Log("Stream started");  
stream.OnStreamStopped += () => Debug.Log("Stream stopped");
```

```
stream.StartStream();  
// ... later ...  
stream.StopStream();
```

Single Emission

Fire one emission without starting continuous streaming:

```
MockStreaming.EmitOnce();  
// or  
MockStream.Instance.EmitOnce();
```

Export from Code

The MockExport static class converts data to formatted strings.

Exporting Series Data

using MockMagic;

```
MockDataPoint[] data = MockSeries.TrendingUp();
```

```
string text = MockExport.Export(data, ExportFormat.Text);  
string csv = MockExport.Export(data, ExportFormat.Csv);  
string json = MockExport.Export(data, ExportFormat.Json);
```



```
string code = MockExport.Export(data, ExportFormat.CSharpCode, variableName:
"sales");
```

Exporting Multi-Series Data

```
MockDataPoint[][] multiData = MockSeries.GenerateMultiSeries(
    SeriesPattern.Seasonal, 3, 12, 0f, 100f, 0.1f, LabelPreset.MonthsShort
);
```

```
string json = MockExport.Export(multiData, ExportFormat.Json);
```

Exporting Simple Data

```
string[] emails = Mock.Emails(20);
string csv = MockExport.Export(emails, ExportFormat.Csv);
```

Copy to Clipboard

```
MockExport.CopyToClipboard(data);           // MockDataPoint[]
MockExport.CopyToClipboard(multiData);      // MockDataPoint[][]
MockExport.CopyToClipboard(emails);         // string[]
MockExport.CopyToClipboard("custom string"); // Raw string
```

Feeding MockMagic Data into Chart Guru

```
using MockMagic;
using ChartGuru;
using UnityEngine;
```

```
public class MockChart : MonoBehaviour
{
    private ChartGraphic _chart;

    private void Start()
    {
        // Generate mock data
        MockDataPoint[] data = MockSeries.Generate(
            SeriesPattern.TrendingUp, 12, 0f, 100f, 0.15f, LabelPreset.MonthsShort
        );

        // Build a Chart Guru chart
        ChartBuilder builder = Chart.Create();
        for (int i = 0; i < data.Length; i++)
        {
```



```
        BarMark bar = new BarMark(i, data[i].Value);
        bar.CategoryLabel = data[i].Label;
        builder.Add(bar);
    }

    Chart chart = builder
        .chartXAxis(x => x.Label("Month"))
        .chartYAxis(y => y.Label("Value"))
        .Build();

    _chart = ChartCanvasHelper.RenderToRectTransform(
        chart, GetComponent<RectTransform>(), true, "MockChart"
    );
}
}
```



FAQ

I WOULD LIKE TO ADD AN ADDITIONAL TYPE

There is no generic concept yet for this but please contact me on Discord for such a request.



SUPPORT

Web: <https://www.wetzold.com/tools>

Discord: <https://discord.gg/uzeHzEMM4B>

