

ChartDirector: Профессиональная библиотека для создания графиков и диаграмм

Что такое ChartDirector?

ChartDirector — это мощная библиотека компонентов для создания профессиональных графиков и диаграмм, поддерживающая множество языков программирования и платформ.

Основные возможности

- **Типы диаграмм:**
 - **Круговые и кольцевые**
 - **Столбчатые и линейные**
 - **Сплайны и ступенчатые линии**
 - **Финансовые графики**
 - **Точечные и пузырьковые**
 - **Контурные и тепловые карты**
 - **Гистограммы и воронки**
 - **Радарные и полярные**
 - **И многие другие**
- **Интерактивность:**
 - **Поддержка масштабирования и прокрутки**
 - **Настраиваемые всплывающие подсказки**
 - **Горячие точки с событиями**
 - **Отслеживание курсора**
- **Производительность:**
 - **Многопоточная архитектура**
 - **Поддержка до 1 миллиарда точек на графике**
 - **Высокая скорость обработки данных**

Форматы вывода

Библиотека поддерживает экспорт графиков в различные форматы:

- **PDF**
- **SVG**
- **PNG**
- **JPG**
- **GIF**
- **BMP**

Поддержка платформ и языков

ChartDirector работает на следующих платформах:

- **Windows**
- **Linux**
- **macOS**
- **FreeBSD**
- **Solaris**

Поддерживает следующие языки программирования:

- **.NET (C#/VB)**
- **Java**
- **C++**
- **ASP**
- **PHP**
- **Perl**
- **Python**

Преимущества использования

- **Универсальность:** подходит для различных типов приложений (веб, десктоп, консольные)
- **Настраиваемость:** гибкая система кастомизации каждого элемента
- **Производительность:** оптимизирована для работы с большими данными

- **Интерактивность:** богатые возможности для пользовательского взаимодействия
- **Масштабируемость:** поддержка многопоточной обработки

Применение

ChartDirector идеально подходит для:

- **Создания отчетов**
- **Визуализации данных**
- **Финансового анализа**
- **Мониторинга в реальном времени**
- **Бизнес-аналитики**
- **Научных исследований**

Во многих примерах ChartDirector в частности в проектах RealTime графиках в исходном коде присутствует довольно распространенный файл `<script src="/js/cdjcvc.js"></script> /js/cdjcvc.js` - что он делает . Зачем он. Повторюсь он используется в Проектах реального времени

Что такое cdjcvc.js в ChartDirector

cdjcvc.js — это специальный JavaScript-файл, который является частью библиотеки ChartDirector и играет важную роль в реализации реального времени (real-time) графиков.

Основные функции файла

1. Клиентская обработка данных:

- **Управление потоками данных в реальном времени**
- **Обработка обновлений графиков без полной перезагрузки страницы**
- **Синхронизация с серверной частью**

2. Интерактивные элементы:

- **Обработка пользовательских событий**
- **Управление всплывающими подсказками**
- **Работа с интерактивными элементами графика**

3. Оптимизация производительности:

- **Буферизация данных**
- **Оптимизация отрисовки**
- **Минимизация нагрузки на браузер**

Как работает cdjcv.js

Механизм работы основан на следующих принципах:

- **Постоянный мониторинг обновлений данных**
- **Автоматическая перерисовка графиков**
- **Оптимизация использования ресурсов браузера**
- **Поддержка различных типов обновлений (полные/частичные)**

Преимущества использования

- **Высокая производительность при работе с данными в реальном времени**
- **Минимальные задержки при обновлении графиков**
- **Оптимизированная загрузка ресурсов**
- **Кроссбраузерная совместимость**

Практическое применение

cdjcv.js активно используется в следующих сценариях:

- **Мониторинг систем в реальном времени**
- **Финансовые графики с обновлениями котировок**
- **Визуализация данных с датчиков IoT**
- **Аналитические панели с динамическими обновлениями**
- **Системы мониторинга производительности**

Рекомендации по использованию

- **Убедитесь, что файл корректно подключен к проекту**
- **Проверьте совместимость с версией ChartDirector**
- **Настройте параметры обновления под конкретные требования проекта**
- **Оптимизируйте частоту обновлений для минимизации нагрузки на систему**

А я думал что это обычный файл , который по-молчанию входит во многие пакеты js. А он оказывается специализированный.

Да, действительно, cdjcv.js — это не универсальный JavaScript-файл, а специализированный компонент, разработанный специально для работы с ChartDirector.

Многие разработчики могут ошибочно принять его за стандартный файл, потому что:

- **Он часто встречается в примерах**
- **Используется в типовых решениях**
- **Интегрирован в демо-версии**

Но на самом деле это ключевой элемент для реализации real-time функциональности в ChartDirector. Без него невозможно обеспечить корректную работу динамических графиков и обновлений в реальном времени.

Важно помнить, что этот файл:

- **Тесно связан с серверной частью ChartDirector**
- **Содержит специфичные функции для работы с графикой**
- **Оптимизирован под конкретные задачи визуализации данных**

Поэтому при разработке проектов с использованием ChartDirector необходимо:

- **Проверять актуальность версии файла**
- **Учитывать его специфику при настройке проекта**
- **Следить за совместимостью с используемой версией библиотеки**

Вот в этом файле Realtimemultichart.cshtml используется cdjcv.js. Расскажи, что в этом коде делают все функции. Расскажи, что делает каждая функция. Трудно разобраться. Я так понял, что это все javascript. @page @model RealtimemultichartModel @{ Layout = null; } <!DOCTYPE html> <html> <head> <title>Real Time MultiChart</title> <script src="/js/cdjcv.js"></script> </head> <body style="margin:0px"> <script type="text/javascript"> // // Execute the following initialization code after the web page is loaded //

```
JsChartViewer.addEventListener(window, 'load', function() { var viewer = JsChartViewer.get('@ViewBag.Viewer.ID'); // Draw track cursor when mouse is moving over plotarea. Hide it when mouse leaves plot area. viewer.attachHandler(["MouseMovePlotArea", "TouchStartPlotArea", "TouchMovePlotArea", "ChartMove", "PostUpdate", "Now"], function(e) { this.preventDefault(e); // Prevent the browser from using touch events for other actions multiTrackLineLabel(viewer, viewer.getPlotAreaMouseX()); }); // When the chart is being updated, by default, an "Updating" box will pop up. In this example, we //
```

```

will disable this box. viewer.updatingMsg = ""; }); // // Draw track line for a MultiChart //
function multiTrackLineLabel(viewer, mouseX) { // Remove all previously drawn
tracking object viewer.hideObj("all"); // Use a loop to draw track labels for the
XYCharts inside the MultiChart for (var i = 0; i < viewer.getChartCount(); ++i) // Only the
bottom chart (i == viewer.getChartCount() - 1) needs x-axis label.
xyChartTrackLabel(viewer, mouseX, viewer.getChart(i), i, i == viewer.getChartCount() -
1); } // // Draw xy chart track label // function xyChartTrackLabel(viewer, mouseX, c, id,
needXAxisLabel) { // The plot area var plotArea = c.getPlotArea(); // The XYChart API
object obtains the various coordinates relative to the top-left corner // of the XYChart.
However, it needs to draw the track cursor on the MultiChart. So we need // to obtain
the coordinates of the XYChart top-left corner inside the MultiChart. var originX =
c.getAbsOffsetX(); var originY = c.getAbsOffsetY(); // Get the data x-value that is
nearest to the mouse, and find its pixel coordinate. var xValue =
c.getNearestXValue(mouseX); var xCoord = originX + c.getXCoord(xValue); // Draw a
vertical track line at the x-position viewer.drawLine("trackLine_" + id, xCoord, originY
+ plotArea.getTopY(), originY + plotArea.getBottomY(), "black 1px dotted"); // Only the
last chart needs to draw the x-axis label if (needXAxisLabel) {
viewer.showTextBox("xAxisLabel_" + id, xCoord, originY + plotArea.getBottomY() + 5,
JsChartViewer.Top, c.xAxis().getFormattedLabel(xValue, "hh:nn:ss"), "font:bold 13px
Arial;color:#FFFFFF;background-color:#000000;padding:0px 3px"); } // Iterate through
all layers to draw the data labels for (var i = 0; i < c.getLayerCount(); ++i) { var layer =
c.getLayerByZ(i); // The data array index of the x-value var xIndex =
layer.getXIndexOf(xValue); // Iterate through all the data sets in the layer for (var j = 0;
j < layer.getDataSetCount(); ++j) { var dataSet = layer.getDataSetByZ(j); // Get the color
and position of the data label var color = dataSet.getDataColor(); var yCoord =
c.getYCoord(dataSet.getPosition(xIndex), dataSet.getUseYAxis()); // Draw a track dot
with a label next to it for visible data points in the plot area if ((yCoord != null) &&
(yCoord >= plotArea.getTopY()) && (yCoord <= plotArea.getBottomY()) && (color != null))
{ viewer.showTextBox("dataPoint_" + id + "_" + i + "_" + j, xCoord, originY + yCoord,
JsChartViewer.Center, viewer.htmlRect(7, 7, color)); viewer.showTextBox("dataLabel"
+ id + "_" + i + "_" + j, xCoord + 5, originY + yCoord, JsChartViewer.Left,
dataSet.getValue(xIndex).toPrecision(4), "padding:0px 3px;font:bold 13px
Arial;background-color:" + color + ";color:#FFFFFF;-webkit-text-size-adjust:100%;"); } } }
} // // Executes once every second to update the countdown display. Updates the chart
when the countdown reaches 0. // function timerTick() { // Get the update period and
the time left var updatePeriod =
parseInt(document.getElementById("UpdatePeriod").value); var timeLeft =
Math.min(parseInt(document.getElementById("TimeRemaining").innerHTML),
updatePeriod) - 1; if (timeLeft == 0) // Can update the chart now
JsChartViewer.get('@ViewBag.Viewer.ID').partialUpdate(); else if (timeLeft < 0) // Reset
the update period timeLeft += updatePeriod; // Update the countdown display
document.getElementById("TimeRemaining").innerHTML = timeLeft; }
window.setInterval("timerTick()", 1000); </script> <table cellpadding="0"
cellpadding="0" border="0"> <tr> <td align="right" colspan="2"
style="background:#000088; color:#ffff00; padding:0px 4px 2px 0px;"> <a
style="color:#FFFF00; font:italic bold 10pt Arial; text-decoration:none"
href="http://www.advsofteng.com/"> Advanced Software Engineering </a> </td> </tr>
<tr valign="top"> <td style="width:150px; background:#c0c0ff; border-right:black 1px
solid; border-bottom:black 1px solid;"> <br /> <br /> <div style="font: 9pt Verdana;

```

```
padding:10px;"]> <b>Update Period</b><br /> <select id="UpdatePeriod"
style="width:130px"> <option value="5" selected="selected">5</option> <option
value="10">10</option> <option value="20">20</option> <option
value="30">30</option> <option value="60">60</option> </select> </div> <div
style="font:9pt Verdana; padding:10px;"]> <b>Time Remaining</b><br /> <div
style="width:128px; border:#888888 1px inset;"]> <div style="margin:3px"
id="TimeRemaining">0</div> </div> </div> </td> <td> <div style="font: bold 20pt Arial;
margin:5px 0px 0px 5px;"]> Real Time MultiChart </div> <hr style="border:solid 1px
#000080" /> <div style="padding:0px 5px 5px 10px"> <!-- ***** Here is the chart
image ***** --> @Html.Raw(ViewBag.Viewer.RenderHTML()) </div> </td> </tr> </table>
</body> </html>
```

Давайте разберем этот код пошагово:

Основная структура

Код представляет собой HTML-страницу с встроенным JavaScript для реализации реального времени графиков.

Основные компоненты

1. Инициализация при загрузке страницы:

javascript

```
JsChartViewer.addEventListener(window, 'load', function() {
    // код инициализации
});
```

Здесь происходит:

- Получение объекта viewer
- Настройка обработчиков событий для отслеживания мыши
- Отключение стандартного сообщения обновления

2. Функция отслеживания курсора:

javascript

```
function multiTrackLineLabel(viewer, mouseX) {
    // код отслеживания
}
```

Эта функция:

- Удаляет старые объекты отслеживания
- Проходит по всем графикам в мультичарте
- Вызывает отрисовку меток для каждого графика

3. Отрисовка меток для XY-графиков:

javascript

```
function xyChartTrackLabel(viewer, mouseX, c, id, needXAxisLabel) {  
    // код отрисовки  
}
```

Здесь происходит:

- Получение области построения графика
- Расчет координат для отслеживания
- Отрисовка вертикальной линии
- Добавление меток значений
- Обработка всех слоев и наборов данных

4. Таймер обновлений:

javascript

```
function timerTick() {  
    // код таймера  
}
```

Этот код:

- Получает период обновления из селекта
- Считает оставшееся время
- Выполняет частичное обновление графика
- Обновляет отображение счетчика

HTML-структура

В HTML части:

- Создается панель управления с настройками обновления
- Отображается сам график через `@Html.Raw(ViewBag.Viewer.RenderHTML())`
- Добавляются элементы управления (выпадающий список периода обновления)

Детальный разбор функций

1. Инициализация:

- `JsChartViewer.get()` - получает объект viewer по ID
- `attachHandler()` - привязывает обработчики событий
- `preventDefault()` - блокирует стандартные действия браузера

2. Отслеживание:

- `getPlotAreaMouseX()` - получает X-координату мыши
- `hideObj()` - скрывает все объекты отслеживания
- `getChartCount()` - получает количество графиков

3. Отрисовка меток:

- `getXCoor()` - получает X-координату по значению
- `drawVLine()` - рисует вертикальную линию отслеживания
- `showTextBox()` - показывает текстовые метки

4. Обновление:

- `partialUpdate()` - выполняет частичное обновление графика
- `setInterval()` - запускает таймер обновления каждую секунду

Практическое применение

Этот код позволяет:

- Отслеживать положение курсора на графике
- Отображать значения в реальном времени
- Настраивать период обновления
- Обновлять график без полной перезагрузки страницы

Все эти функции работают вместе, создавая интерактивный график с возможностью отслеживания данных в реальном времени.

В этом коде используется partial view ?

В данном коде явное использование Partial View (частичного представления) не наблюдается. Давайте разберем структуру:

1. Это целая страница (определяется по тегам `<html>`, `<head>`, `<body>`)
2. Используется синтаксис Razor (`@page`, `@model`, `@ViewBag`)
3. Основной вывод графика происходит через:

`csharp`

```
@Html.Raw(ViewBag.Viewer.RenderHTML())
```

Это не partial view, а:

- Razor Page (определяется по `@page` директиве)

- Самодостаточная страница с полной HTML-структурой
- Использует ViewBag для передачи данных

Признаки того, что это не partial view:

- Наличие полного HTML-документа
- Отсутствие директивы `@{ Layout = "_Layout"; }` (хотя она переопределена как null)
- Самостоятельный жизненный цикл страницы

Если бы это был partial view, мы бы увидели:

- Отсутствие тегов `<html>`, `<head>`, `<body>`
- Использование в другом представлении через `@await Html.PartialAsync()`
- Отсутствие директивы `@page`

Таким образом, данный код представляет собой полноценную Razor Page, а не partial view.

Тогда объясни как средствами javascript происходит Rendering страницы Html объектами (сущностями) языка C#. То есть здесь получается, что функции javascript работают с данными C#. Вот второй код C#.

```
public class RealtimeMultichartModel : PageModel { // // Draw a single chart // private XYChart
drawXYChart(RazorChartViewer viewer, DateTime[] timeStamps, double[] dataSeries,
string name, int color, Axis xAxisScale, bool xAxisVisible) { // Only the last chart has
an x-axis int xAxisHeight = 25; if (!xAxisVisible) { xAxisHeight = 0; } // Create an
XYChart object of size 640 x 120 pixels (excluding x-axis height) XYChart c = new
XYChart(640, 120 + xAxisHeight); // Set the plotarea at (55, 10) with width 85 pixels
less than chart width, and height // 20 pixels less than chart height. Use a vertical
gradient from light blue (f0f6ff) to // sky blue (a0c0ff) as background. Set border to
transparent and grid lines to white // (ffffff). c.setPlotArea(55, 10, c.getWidth() - 85,
c.getHeight() - 20 - xAxisHeight, c.linearGradientColor(0, 10, 0, c.getHeight() - 20 -
xAxisHeight, 0xf0f6ff, 0xa0c0ff ), -1, Chart.Transparent, 0xffffffff, 0xffffffff); // As the data
can lie outside the plotarea in a zoomed chart, we need enable clipping.
c.setClipping(); // Add a legend box at (55, 5) using horizontal layout. Use 8pts Arial
Bold as font. Set // the background and border color to Transparent and use line style
legend key. LegendBox b = c.addLegend(55, 5, false, "Arial Bold", 10);
b.setBackground(Chart.Transparent); b.setLineStyleKey(); // Set the x and y axis stems
to transparent and the label font to 10pt Arial c.xAxis().setColors(Chart.Transparent);
c.yAxis().setColors(Chart.Transparent); c.xAxis().setLabelStyle("Arial", 10);
c.yAxis().setLabelStyle("Arial", 10); // Add axis title using 10pts Arial Bold Italic font
c.yAxis().setTitle(name, "Arial Bold", 10);
//=====
===== // Add data to chart
//=====
===== // Add a line layer with the given data, with a line width of 2 pixels.
```

```

LineLayer layer = c.addLineLayer(); layer.setLineWidth(2);
layer.setXData(timeStamps); layer.addDataSet(dataSeries, color, name);
//=====
===== // Configure axis scale and labelling
//=====
===== // For the automatic axis labels, set the minimum spacing to 30 pixels for
the y axis. c.yAxis().setTickDensity(30); if (xAxisScale == null) { // If xAxisScale is
given, then use it to synchronize with other charts. c.xAxis().copyAxis(xAxisScale); }
else { // Set the x-axis label format c.xAxis().setLabelFormat("{value|hh:nn:ss}"); } //
Hide the x-axis if it is not visible. if (!xAxisVisible) {
c.xAxis().setColors(Chart.Transparent, Chart.Transparent); }
//=====
===== // Output the chart
//=====
===== return c; } // // Draw the MultiChart // private void
drawChart(RazorChartViewer viewer) { // // Data to draw the chart. In this demo, the
data buffer will be filled by a random data // generator. In real life, the data is probably
stored in a buffer (eg. a database // table, a text file, or some global memory) and
updated by other means. // // We use a data buffer to emulate the last 240 samples. int
sampleSize = 240; double[] dataSeries1 = new double[sampleSize]; double[]
dataSeries2 = new double[sampleSize]; double[] dataSeries3 = new
double[sampleSize]; DateTime[] timeStamps = new DateTime[sampleSize]; // Our
pseudo random number generator DateTime firstDate = DateTime.Now.AddSeconds(-
timeStamps.Length); for(int i = 0; i < timeStamps.Length; ++i) { timeStamps[i] =
firstDate.AddSeconds(i); double p = timeStamps[i].Ticks / 10000000; dataSeries1[i] =
Math.Cos(p * 2.1) * 10 + 1 / (Math.Cos(p) * Math.Cos(p) + 0.01) + 20; dataSeries2[i] =
100 * Math.Sin(p / 27.7) * Math.Sin(p / 10.1) + 150; dataSeries3[i] = 100 * Math.Cos(p /
6.7) * Math.Cos(p / 11.9) + 150; } // The MultiChart, initially set to a height 10 pixels as
the top margin. MultiChart m = new MultiChart(640, 10); // This first chart is
responsible for setting up the x-axis scale. XYChart xyc = drawXYChart(viewer,
timeStamps, dataSeries1, "Alpha", 0xff0000, null, false); Axis xAxisScale =
xyc.xAxis(); // Add the XYChart to the MultiChart and increase the MultiChart height
m.addChart(0, m.getHeight(), xyc); m.setSize(m.getWidth(), m.getHeight() +
xyc.getHeight()); // All other charts synchronize their x-axes with that of the first chart.
xyc = drawXYChart(viewer, timeStamps, dataSeries2, "Beta", 0x008800, xAxisScale,
false); // Add the XYChart to the MultiChart and increase the MultiChart height
m.addChart(0, m.getHeight(), xyc); m.setSize(m.getWidth(), m.getHeight() +
xyc.getHeight()); // The last chart displays the x-axis. xyc = drawXYChart(viewer,
timeStamps, dataSeries3, "Gamma", 0x0000ff, xAxisScale, true); // Add the XYChart to
the MultiChart and increase the MultiChart height m.addChart(0, m.getHeight(), xyc);
m.setSize(m.getWidth(), m.getHeight() + xyc.getHeight()); // Set the combined plot area
to be the bounding box of the plot areas of the 3 charts m.setMainChart(m); // Output
the chart viewer.Image = m.makeWebImage(Chart.SVG); // Output Javascript chart
model to the browser to support tracking cursor viewer.ChartModel =
m.getJsChartModel(); } public IActionResult OnGet() { RazorChartViewer viewer = new
RazorChartViewer(HttpContext, "chart1"); ViewData["Viewer"] = viewer; // // This script
handles both the full page request, as well as the subsequent partial // updates (AJAX
chart updates). We need to determine the type of request first before // we processing
it. // if (RazorChartViewer.IsPartialUpdateRequest(Request)) { // Is a partial update

```

`request. drawChart(viewer); return Content(viewer.PartialUpdateChart()); } // // If the code reaches here, it is a full page request. // drawChart(viewer); return Page(); } }`

Давайте разберем, как происходит взаимодействие между JavaScript и C# в данном приложении:

Архитектура взаимодействия

1. C# (Backend):

- Создает и настраивает графики
- Генерирует HTML-изображение графика
- Передает данные в JavaScript через `ViewBag`/`ViewData`

2. JavaScript (Frontend):

- Обработывает пользовательские события
- Управляет интерактивностью
- Обновляет части страницы

Процесс рендеринга

1. Инициализация на C#:

csharp

```
public IActionResult OnGet()
{
    RazorChartViewer viewer = new RazorChartViewer(HttpContext, "chart1");
    ViewData["Viewer"] = viewer;

    // Логика отрисовки
    drawChart(viewer);
    return Page();
}
```

Здесь:

- Создается объект `RazorChartViewer`
- Данные передаются в представление через `ViewData`
- Вызывается метод отрисовки

2. Передача данных в JavaScript:

javascript

```
JsChartViewer.get('@ViewBag.Viewer.ID')
```

- Через Razor-синтаксис (`@ViewBag`) данные C# передаются в JavaScript

- JavaScript получает доступ к ID viewer для управления графиком

Механизм обновления

1. Частичное обновление:

csharp

```
if (RazorChartViewer.IsPartialUpdateRequest(Request)) {  
    drawChart(viewer);  
    return Content(viewer.PartialUpdateChart());  
}
```

- Проверяется тип запроса
- При частичном обновлении вызывается **PartialUpdateChart()**

2. JavaScript-обработчики:

javascript

```
viewer.attachHandler(["MouseMovePlotArea", "TouchStartPlotArea",  
"TouchMovePlotArea", "ChartMove",  
"PostUpdate", "Now"], function(e) {  
    this.preventDefault(e);  
    multiTrackLineLabel(viewer, viewer.getPlotAreaMouseX());  
});
```

- JavaScript обрабатывает события
- Вызывает функции C# через API viewer

Взаимодействие компонентов

1. JavaScript управляет:

- Отслеживанием курсора
- Обработкой событий
- Визуальными эффектами

2. C# управляет:

- Генерацией данных
- Созданием графиков
- Логикой отрисовки

Пример взаимодействия

1. Создание графика на C#:

csharp

```
viewer.Image = m.makeWebImage(Chart.SVG);
viewer.ChartModel = m.getJsChartModel();
```

- Создается изображение графика
- Передается модель для JavaScript

2. Использование в JavaScript:

javascript

```
function multiTrackLineLabel(viewer, mouseX) {
    // Работа с данными, полученными от C#
    viewer.hideObj("all");
    // ...
}
```

Итоговый процесс

1. C# генерирует график и передает его в JavaScript
2. JavaScript управляет интерактивностью и обновлениями
3. Данные передаются через:
 - ViewBag / ViewData
 - API viewer
 - JavaScript-функции

Таким образом, создается единый механизм, где C# отвечает за генерацию и логику, а JavaScript — за интерактивность и пользовательский интерфейс.

То есть здесь обновление partial view все же присутствует ?

Да, вы правы, частичное обновление (partial update) действительно присутствует, но это не Partial View в классическом понимании Razor Pages/MVC.

Это частичное обновление (AJAX-обновление) части страницы:

1. Механизм работает через:
 - AJAX-запросы
 - Метод PartialUpdateChart()
 - Специальную проверку RazorChartViewer.IsPartialUpdateRequest(Request)
2. Процесс обновления:

csharp

```
if (RazorChartViewer.IsPartialUpdateRequest(Request)) {
    drawChart(viewer);
    return Content(viewer.PartialUpdateChart());
}
```

```
}
```

Здесь:

- Проверяется тип запроса
- При частичном обновлении вызывается `PartialUpdateChart()`
- Возвращается только обновленная часть данных

3. Отличие от Partial View:

- Нет отдельного файла представления
- Обновляется только часть данных
- Работает через AJAX
- Не перезагружает всю страницу

4. Как это работает:

- JavaScript отправляет AJAX-запрос
- C# обрабатывает запрос
- Возвращает только обновленные данные графика
- JavaScript применяет эти обновления к существующей странице

Таким образом, это не **Partial View** (отдельный файл представления), а частичное обновление существующего представления через AJAX-механизм, что позволяет обновлять только необходимые части страницы без полной перезагрузки.

Тогда приведи мне простой пример Классического **PartialView** (**Partial View** в классическом понимании **Razor Pages/MVC**)

Давайте рассмотрим классический пример использования **Partial View** в **ASP.NET Core MVC/Razor Pages**.

Структура проекта

1. **Main View (Index.cshtml)**
2. **Partial View (_ChartPartial.cshtml)**
3. **Controller/PageModel**

Пример реализации

1. **Partial View (_ChartPartial.cshtml)**
cshtml

```
@model ChartModel
```

2. Главная View (Index.cshtml)

```
cshtml
```

```
@page
```

```
@model IndexModel
```

Главная страница

```
@await Html.PartialAsync("_ChartPartial", Model.ChartData)
```

3. PageModel (Index.cshtml.cs)

```
csharp
```

```
public class IndexModel : PageModel
```

```
{
```

```
    public ChartModel ChartData { get; set; }
```

```
    public void OnGet()
```

```
    {
```

```
        // Инициализация данных для Partial View
```

```
        ChartData = new ChartModel
```

```
        {
```

```
            ChartImage = GenerateChartImage(),
```

```
            // Другие параметры
```

```
        };
```



```

}

public IActionResult OnPostUpdateChart()
{
    // Генерация новых данных для Partial View
    ChartData = new ChartModel
    {
        ChartImage = GenerateChartImage(),
        // Другие параметры
    };

    return PartialView("_ChartPartial", ChartData);
}

private string GenerateChartImage()
{
    // Логика генерации изображения графика
    return "path_to_chart_image";
}
}

```

Объяснение работы

1. Partial View (_ChartPartial.cshtml):

- Содержит только код, относящийся к графику
- Принимает модель `ChartModel`
- Может быть повторно использован в разных представлениях

2. Главная View (Index.cshtml):

- Вызывает Partial View через `@await Html.PartialAsync()`
- Содержит основной контент страницы
- Может включать несколько Partial View

3. PageModel:

- Содержит логику для главной страницы
- Имеет отдельный метод для обновления Partial View
- Управляет данными, передаваемыми в Partial View

Преимущества такого подхода

- Повторное использование кода

- **Чистота структуры проекта**
- **Легкость поддержки**
- **Модульность кода**
- **Разделение ответственности**

Этот пример демонстрирует классический подход к использованию Partial View в ASP.NET Core, где Partial View представляет собой отдельный файл с изолированной функциональностью, который может быть включен в любое другое представление.