# sgcHTML

## Getting Started

Build your first Delphi web application, one small demo at a time

For sgcWebSockets 2026.10 or later

eSeGeCe · October 2026

eSeGeCe SOFTWARE

# About this guide

This guide is for Delphi developers who know how to build a VCL form and want to put that form on the web with sgcHTML. You do not need to know HTML, CSS or JavaScript: every page in the guide is designed on a Delphi form, with components you drop and configure as usual.

The guide follows the six QuickStart demos. Each demo adds one idea to the previous one, from a single card to a small site with several pages and a database. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, look at the page in the browser, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcHTML is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\60.HTML\00.QuickStart` folder of your sgcWebSockets installation. Each demo has its own subfolder, from `01.HelloWorld` to `06.Pages`, with a Delphi project ( `.dpr` ) that you can open and run directly.

# Contents


## 1. What sgcHTML is

The idea in one minute
Your first page needs one line of code
How a request flows
What you need
Coming from uniGUI or the VCL

## 2. Install and the palette

Editions and the trial
Installing
The palette pages
The Web Design tab
Preview in Browser and Design in Browser
The CRUD wizard
Where the demos are
Running your first demo

## 3. Core concepts

The form is the page
The four building blocks
One form, one URL
Actions are your events
Threads and the lock
One page for every visitor
Encode what users type
Data binding
Everything is inside the executable
Summary: who does what on a click

## 4. The QuickStart demos

4.1 Demo 01: Hello World
4.2 Demo 02: Components
4.3 Demo 03: Events
4.4 Demo 04: Forms
4.5 Demo 05: Database
4.6 Demo 06: Pages

## 5. Main components

Reference table
Layout
Data
Feedback
Input
Security
Navigation

## 6. Common problems

Setup and serving
Events and threads
Forms
Layout and appearance

## 7. Where next

The design-time demos
The run-time demos
The code-first path
Help topics
On the website
Getting help

# 1. What sgcHTML is

sgcHTML is a set of server-side HTML user interface components for Delphi. You build a web page the way you build a desktop window: you drop components on a form, set their properties in the Object Inspector and write event handlers in Pascal. Your application then serves that page to any web browser.

The browser receives ordinary Bootstrap 5 markup, so the page looks like a modern web application on a desktop, a tablet or a phone. When the user clicks a button or submits a form, a small library called htmx sends the request to your application and swaps the answer into the page. You write no JavaScript, no HTML templates and no request handlers to get there.

This guide is for Delphi developers who are new to sgcHTML. You know Delphi well, you may know a little HTML, and you may come from uniGUI or from plain VCL desktop work. By the end you will know how the pieces fit together, which components to reach for first, and how to avoid the mistakes that most people make in their first hour.

## The idea in one minute

Everything runs in your Delphi program. The browser is only a display.

- **Your Delphi code renders the page on the server.** Each sgcHTML component knows how to write itself as Bootstrap markup. A `TsgcHTMLComponent_Panel` becomes a Bootstrap card, a `TsgcHTMLComponent_Grid` becomes a table, a `TsgcHTMLComponent_Chart` becomes a Chart.js chart.
- **The browser shows Bootstrap.** Styling, responsive columns and dialogs come from Bootstrap 5, which is already included.
- **htmx swaps parts of the page.** A click does not reload the whole page. Only the component you named as the target is drawn again on the server and replaced in the browser.
- **No JavaScript to write.** The scripts the page needs (htmx, Bootstrap and Chart.js) are compiled into your executable and served by sgcHTML itself.

## Your first page needs one line of code

Starting with sgcWebSockets 2026.10 there is a zero-code path. You drop four kinds of components on a VCL form:

1. a `TsgcHTTPServer`, the web server, with its `Port` set;
2. a `TsgcHTMLDesignServer`, which connects the web server to the page: set its `Server` to the HTTP server, its `Surface` to the surface and its `PagePath` to the URL of the page (`'/'` by default);
3. a `TsgcHTMLDesignSurface`, the page itself, with its `Title`;
4. the sgcHTML components that make up the page, for example a `TsgcHTMLComponent_Panel`.

Then you start the server. This is the complete event handler of Demo 01, Hello World:

```
procedure TfrmHelloWorld.FormCreate(Sender: TObject);
begin
  { Start listening. From now on http://127.0.0.1:8081/ shows the page. }
  HTTPServer.Active := True;
end;
```

Open `http://127.0.0.1:8081/` in a browser and the panel you designed on the form appears as a Bootstrap card.

Demo 01 in the browser. The card is a TsgcHTMLComponent_Panel dropped on a Delphi form.

> **Note.** The QuickStart demos described in this guide need sgcWebSockets 2026.10 or later. Older versions have the components but not the zero-code path, where the DesignServer builds the page straight from the form.

## How a request flows

It helps to know what happens behind the scenes, because it explains most of the rules in the later chapters.

### When the browser opens the page

1. The browser asks for a URL, for example `http://127.0.0.1:8081/`.
2. `TsgcHTTPServer` receives the request on one of its server threads.
3. The `TsgcHTMLDesignServer` whose `PagePath` matches the path takes the request.
4. The `TsgcHTMLDesignSurface` reads the sgcHTML components on its owner form and renders them. Their positions and widths on the form become Bootstrap rows and columns.
5. The complete HTML page goes back to the browser. The browser then asks for the files the page links: `/bootstrap.min.css`, `/bootstrap.bundle.min.js`, `/htmx.min.js` and, when the page has a chart, `/chart.umd.min.js`. The DesignServer answers those requests too, from resources inside your executable.

### When the user clicks a button

1. htmx sends a request to an action route that the Surface generated for the button.
2. The DesignServer finds the matching item in `Surface.Actions` and calls its `OnExecute` handler, your Pascal code.
3. Your handler changes component properties, for example `ResultPanel.Body`.
4. The DesignServer renders the action's `Target` component again and sends only that piece of HTML back.
5. htmx replaces the old component in the page with the new one. No page reload.

Requests that nobody serves, such as a mistyped URL, get a `404 Not Found` answer.

## What you need

- **Delphi with sgcHTML installed.** sgcHTML is part of sgcWebSockets All-Access and is also sold on its own as the sgcHTML pack. The All-Access trial includes it. Chapter 2 covers the installation.

- **A VCL application on Windows** for the zero-code path. On Windows each sgcHTML component is a real control, which is what lets the DesignServer read the layout from the form. sgcHTML is not available on Android and iOS targets.
- **A web browser.** Any current browser works, on the same machine or on another one in your network. Nothing is installed on the client.
- **No internet connection.** Bootstrap, htmx and Chart.js are served from your executable, not from a CDN, so the page also works on an isolated intranet.
- **Optional: Delphi 10.4 or later** with the WebView2 runtime, for the Web Design tab that previews the page inside the IDE. The other design tools work from Delphi 7 up.

## Coming from uniGUI or the VCL

Many ideas carry over directly. A few do not, and it is better to know them before you write your first handler. The table maps what you know to what sgcHTML uses.

| You know (VCL or uniGUI) | In sgcHTML | Difference to keep in mind |
|---|---|---|
| A form ( `TForm` , `TUniForm` ) | A `TForm` with a `TsgcHTMLDesignSurface` and a `TsgcHTMLDesignServer` . One form is one page at its `PagePath` . | The form itself is never shown to web users. Only its sgcHTML components become the page. |
| `Button1Click` , `TUniButton.OnClick` | An item in `Surface.Actions` : `Source` is the button, `Target` is the component to redraw, `OnExecute` is your code. | The handler runs on a server thread, not the main thread. You say which component is redrawn afterwards. |
| `TEdit` , `TMemo` , `TComboBox` | `TsgcHTMLComponent_Edit` , `TsgcHTMLComponent_Memo` , `TsgcHTMLComponent_Select` , or the fields of a `TsgcHTMLComponent_Form` . | The fields of a submitted `TsgcHTMLComponent_Form` reach your code in `aAction.Params` , by field name. |
| `TDBGrid` , `TUniDBGrid` | `TsgcHTMLComponent_Grid` with `DataSource` and `DataAutoRefresh` . | The grid reloads from the dataset when it changes, much like a `TDBGrid` . |
| `TPageControl` | `TsgcHTMLComponent_Tabs` | The tab pages are items with a `Title` and a `Content` , not child controls. |
| ServerModule (port, files, settings) | `TsgcHTTPServer` (port and server options) plus the DesignServer, which serves the page and its script and style files. | There is no separate server module unit. The server is an ordinary component on a form. |
| MainModule, one per session | No direct counterpart in the zero-code path. | **One designed form is one page shared by every browser.** uniGUI creates forms per session, sgcHTML does not. See chapter 3. |
| Data modules | Ordinary `TDataModule` s, or the data components on the page form, as in Demo 05. | Action handlers run one at a time, so a single connection is safe for a QuickStart application. |
| Ajax events, adding JavaScript to a session | The htmx swap of the action's `Target` . | You do not write JavaScript. |

> **Watch out.** The biggest difference from uniGUI is state. In uniGUI every user gets their own copy of each form. In the sgcHTML zero-code path there is one form for the whole application, so a counter or a result panel that one user changes is what every other user sees when they reload. That is fine for dashboards, kiosks, shared status pages and internal tools used by one person at a time. Per-user data needs sessions, which chapter 3 explains.

The other big difference is threading. In a desktop application your `OnClick` runs on the main thread. In sgcHTML your `OnExecute` runs on a server thread, and it must leave the VCL controls of the desktop window alone. Chapter 3 shows what you can safely touch.

# 2. Install and the palette

This chapter gets sgcHTML into your IDE, shows you where the components live, and introduces the design tools that let you see the web page while you build it.

## Editions and the trial

sgcHTML ships in two ways:

- **Inside sgcWebSockets All-Access.** The All-Access edition compiles the HTML pack in, which turns on the `SGC_HTML` and `SGC_HTMX` features.
- **As the sgcHTML pack**, sold on its own.

The All-Access trial includes sgcHTML, so you can follow this whole guide before you buy anything. The install has the same shape in every case.

> **Note.** sgcHTML is not compiled for Android and iOS targets. If the palette page is missing while one of those platforms is selected in the Projects window, that is expected. Switch the target platform to Windows.

## Installing

In the steps below, `{$DIR}` stands for the folder where you unzip the download.

1. **Unzip** the download to a folder of your choice.
2. **Set the library path.** Open *Tools → Options → Library* and add `{$DIR}\source` and the lib folder for your IDE, for example `{$DIR}\libD13\$(Platform)`.
3. **Build the packages.** Open the package group for your IDE version under `{$DIR}\Packages\`. Compile the runtime package first, then install the design-time package, the one whose name starts with `dcl`.
4. **Check the palette.** The pages described below appear in the Tool Palette.
5. **Run a QuickStart demo**, as described at the end of this chapter.

## The palette pages

Three palette pages matter for this guide.

| Palette page | What you take from it |
|---|---|
| **SGC HTTP** | `TsgcHTTPServer`, the web server every sgcHTML application needs. It is included with sgcHTML. |
| **SGC HTML Designer** | The two components of the zero-code path: `TsgcHTMLDesignSurface` (the page) and `TsgcHTMLDesignServer` (serves the page). |
| **SGC HTML** | The page components: 128 classes, from panels, grids and charts to forms, menus and dialogs. Chapter 5 introduces the ones to learn first. |

Dropping a component from the SGC HTML page on a form adds its unit to the `uses` clause, as with any Delphi component. Most components live in a unit named after the class, for example `TsgcHTMLComponent_Panel` in `sgcHTML_Component_Panel`. A few share a unit: `TsgcHTMLComponent_Memo` is in `sgcHTML_Component_Edit`. The two designer components live in `sgcHTMLDesign_Surface` and `sgcHTMLDesign_Server_Indy`.

## The Web Design tab

On Delphi 10.4 and later, a form that holds a `TsgcHTMLDesignSurface` gets an extra editor tab called **Web Design**, next to *Code* and *Design*. It shows a live preview of the web page, rendered from the components on the form.

The tab has a small toolbar:

- **Refresh** reads the form again and renders the page. The tab does not watch the form on its own, so click Refresh after you move or change components.
- **Preview in browser** and **Design in browser** do the same as the context menu items described next.
- **A device width selector** renders the page at Desktop, Tablet or Phone width, so you see how the Bootstrap columns wrap on a narrow screen.

The embedded preview uses the Microsoft WebView2 runtime. If WebView2 cannot start on your machine, the two browser buttons still work, because they open your default browser instead.

## Preview in Browser and Design in Browser

Right-click the `TsgcHTMLDesignSurface` on the form. Its context menu has two items, available in every Delphi version from 7 to 13:

- **Preview in Browser...** writes a read-only snapshot of the page to a temporary file, fills empty data components with sample rows, and opens it in your default browser. Use it to show the page to someone or to check the layout.
- **Design in Browser...** opens a live design canvas in your browser. Components you add, move or remove there are written straight back onto the form in the IDE, and the IDE marks the form as modified. The canvas is served by a small listener bound to `127.0.0.1`, so only your own machine can reach it.

The canvas always rebuilds itself from what is on the form when you open it. A row or column you add only with the canvas's layout tools, with no component inside, is gone the next time you open it. Components are never lost this way, because the canvas creates them on the form the moment you add them.

> **Watch out.** `TsgcHTMLDesignServer` also has an `Enabled` property that turns on the same designer routes inside your running application. It is `False` by default and should stay that way in anything you deploy, because the designer routes are not authenticated. The page at `PagePath` is served whether `Enabled` is on or off.

## The CRUD wizard

On Delphi 10.4 and later the *Tools* menu has an item called **New sgcHTML CRUD page**. It works on the form that is open in the designer: it lists the datasets already on that form, reads the fields of the one you pick, and drops a `TsgcHTMLComponent_CRUD` configured from them, columns included.

The wizard writes no code. It puts the two lines that wire the component to the HTMX engine on the clipboard, for you to paste where your application handles its messages. That wiring belongs to the code-first path, described in the help topic `HTML_CRUD.htm`. For your first database page, the grid and forms approach of Demo 05 is simpler.

## Where the demos are

All sgcHTML demos are under `Demos\60.HTML` in the installation folder. Work through them in this order:

1. **`00.QuickStart`**: six small VCL projects that use the zero-code path, one idea each. Chapter 4 walks through all of them.
2. **`02.DesignTime`**: 22 larger applications designed on forms, such as an ERP, a point of sale and a helpdesk.
3. **`01.RunTime`**: 22 applications with the same names, where the pages are built in code instead.

The folder also holds `10.Lazarus` and `20.Service`, for later.

| QuickStart demo | Port | What it teaches |
|---|---|---|
| 01.HelloWorld | 8081 | The four building blocks and the single line of code. |
| 02.Components | 8082 | A dashboard from the main components, and how the form layout becomes Bootstrap rows and columns. |
| 03.Events | 8083 | Actions: a click in the browser runs `OnExecute` on the server. |
| 04.Forms | 8084 | Submitting a form and reading the typed values. |
| 05.Database | 8085 | A grid bound to FireDAC, with add, edit and delete. |
| 06.Pages | 8086 | Several pages, one form each, sharing one web server and one menu. |

## Running your first demo

1. Open `Demos\60.HTML\00.QuickStart\01.HelloWorld\HelloWorld.dpr`.
2. Run it with F9. A small desktop window appears with the address of the page and an **Open in browser** button.
3. Click the button, or type `http://127.0.0.1:8081/` into any browser on the same machine.

> **Tip.** Each QuickStart demo uses its own port, so you can run all six at the same time and compare them side by side in browser tabs.

The first time a program listens on a port, Windows Firewall may ask whether it may accept network connections. A browser on the same machine reaches `127.0.0.1` either way. Allow access only if other computers in your network should open the page, and then use the machine's network address instead of `127.0.0.1`.

# 3. Core concepts

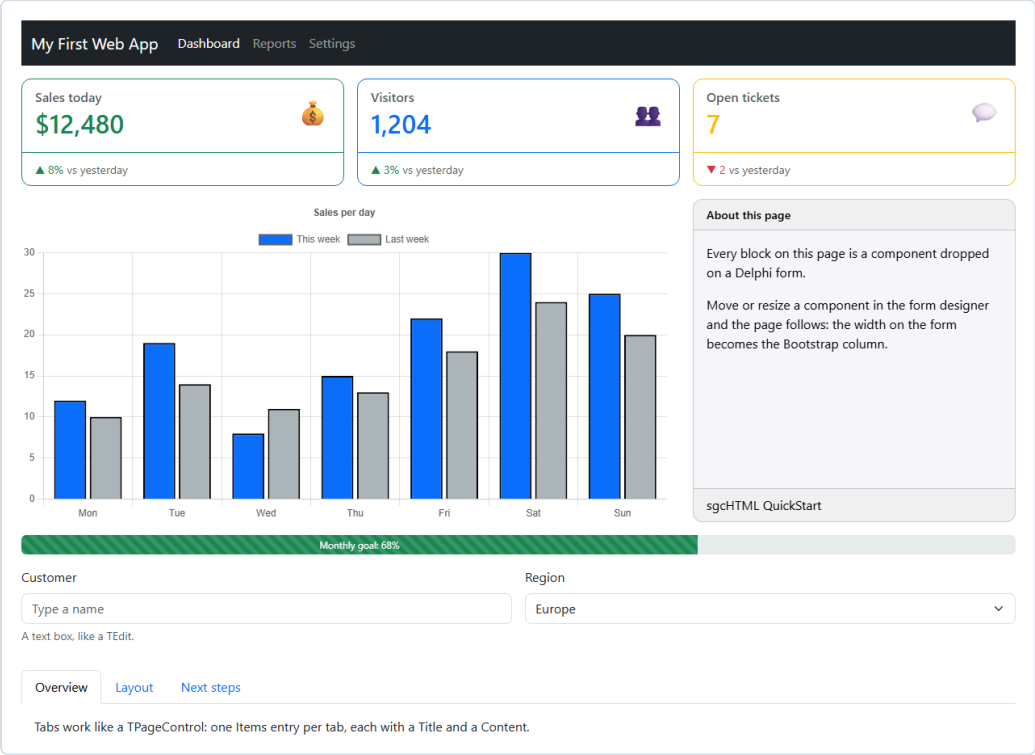
Six ideas cover almost everything you do with the zero-code path: the form is the page, four building blocks serve it, each form has its own URL, actions are the events, handlers run on server threads with one shared page, and data components bind to a `TDataSource`. This chapter explains each one, with the reasons behind the rules.

## The form is the page

The `TsgcHTMLDesignSurface` builds the page from the sgcHTML components on its owner form. You lay out the page in the ordinary form designer, and the Surface reads the layout back as Bootstrap rows and columns. Bootstrap divides every row into 12 columns, and the Surface maps your form onto that grid with these rules:

| On the form | On the page |
|---|---|
| **Top** | Components whose tops are within 12 pixels of each other share one row. Rows follow each other from top to bottom. |
| **Left** | Orders the components inside their row, left to right. |
| **Width** | Becomes the column width: the component's width times 12, divided by the form's client width, rounded, and kept between 1 and 12. A `ColumnWidth` you set yourself wins over this calculation. |
| **Parent** | A component dropped inside another sgcHTML component is nested inside it on the page. Its width is then measured against the width of that parent component. |
| **Left of a lone component** | A row with a single component that does not start at the left edge gets a Bootstrap offset, so it keeps its place. |

A worked example from Demo 02: the form is 960 pixels wide, so one Bootstrap column is 80 pixels. The three stat cards are 312 pixels wide each, which rounds to 4 columns. The chart is 636 pixels wide and becomes 8 columns, and the panel next to it takes the remaining 4. The search box and the drop-down list are 474 pixels each, which is 6 columns, half a row.




Demo 02 in the browser. Each block is a component on the form, and the widths on the form became Bootstrap columns.

Three more details complete the picture:

- **Only sgcHTML components are part of the page.** Ordinary VCL controls on the same form, such as a `TLabel` or a `TButton`, are ignored by the web page. The QuickStart demos use them for the desktop window: a label with the URL and an *Open in browser* button.
- **Frames work.** Components on a `TFrame` that sits on the form are part of the page. Demo 06 designs its menu once on a frame and drops that frame on every page form.
- **`Surface.CompactRows`** removes the row and column wrapper around a component that is alone in its row and already spans the full width. Turn it on when a full-width navigation bar or progress bar shows extra space around it. It is off by default.

## The four building blocks

Every page in the zero-code path is made of the same four parts. Hello World uses exactly these and nothing else.

| Component | Role | Properties you set |
|---|---|---|
| `TsgcHTTPServer` | The web server. It listens on a port and answers the browser. | `Port`. You set `Active` to `True` in code. |
| `TsgcHTMLDesignServer` | Connects the web server to one page. It serves the page at `PagePath`, the script and style files the page links, and the action requests. | `Server`, `Surface`, `PagePath` |
| `TsgcHTMLDesignSurface` | The page. It reads the form layout and renders the HTML, and it holds the `Actions`. | `Title`, `Theme`, `CompactRows`, `Actions` |
| sgcHTML components | The content of the page: panels, grids, charts, forms and so on. | Their own properties, in the Object Inspector. |

The Surface's `Theme` property selects the color scheme of the page: `ptDefault`, `ptDark` or `ptDarkPro`.

## One form, one URL

`PagePath` is the URL path of the page, `'/'` by default. To build a site with several pages, give each page its own form with its own Surface and DesignServer, and give each DesignServer its own `PagePath`. All the DesignServers can share one `TsgcHTTPServer`: each one answers only its own path and leaves the others alone.

In Demo 06 the main form owns the web server and serves `'/'`. Two other forms serve `'/customers'` and `'/about'`, and their DesignServers point at `frmMain.HTTPServer`. The project file creates the main form first, then the page forms:

```
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  { The main form first: it owns the web server the other pages use. }
  Application.CreateForm(TfrmMain, frmMain);
  { The page forms. They stay hidden, they only exist to be served. }
  Application.CreateForm(TfrmCustomers, frmCustomers);
  Application.CreateForm(TfrmAbout, frmAbout);
  Application.Run;
```

Each DesignServer plugs itself into the web server while its form is created. Start the server only after all page forms exist. Demo 06 does this in the main form's `OnShow`, which runs once `Application.Run` starts. Chapter 6 shows what happens when you start too early.

Links between pages are plain URLs. A `TsgcHTMLComponent_NavBar` item with `Href` set to `/customers` opens that page.

## Actions are your events

In a desktop application you double-click a button and write `Button1Click`. In sgcHTML the same job is done by an item in the Surface's `Actions` collection, of class `TsgcHTMLDesignAction`. Open the collection editor from the Object Inspector, add an item and set:

| Property | Meaning |
|---|---|
| `ActionName` | A name for the action, for your own reference. |
| `Source` | The component the user acts on: a button, or a `TsgcHTMLComponent_Form` for a submit. |
| `Target` | The component that is rendered again and swapped into the page after your handler runs. |
| `Trigger` | What starts the action. `dtClick` is the default. Use `dtSubmit` for forms. The enumeration also has `dtChange`, `dtLoad`, `dtRevealed` and `dtInterval`, which the QuickStart does not use. |
| `Confirm` | Optional. When set, the browser asks this question first, and the action runs only if the user confirms. |
| `OnExecute` | Your Pascal code. Double-click the event in the Object Inspector to create it. |

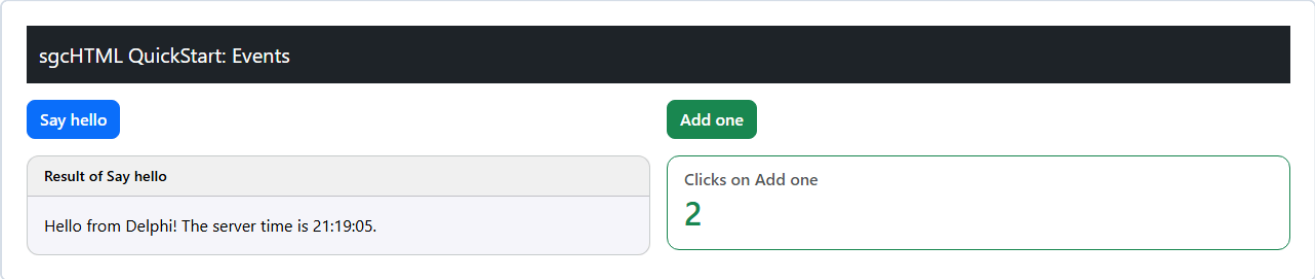
This is how Demo 03 stores its two actions in the `.dfm`, as you see it with *View as Text*:

```
    Actions = <
      item
        ActionName = 'SayHello'
        Source = HelloButton
        Target = ResultPanel
        OnExecute = HelloActionExecute
      end
      item
        ActionName = 'AddOne'
        Source = AddButton
        Target = ClicksCard
        OnExecute = AddActionExecute
      end>
```

And this is the first handler. It changes one property of the target, and that is all:

```
procedure TfrmEvents.HelloActionExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  ResultPanel.Body := 'Hello from Delphi! The server time is ' +
    TimeToStr(Now) + '.';
end;
```

When the handler returns with `aResponse` left empty, the DesignServer renders the `Target` again and sends it to the browser, which swaps it in place. Leaving `aResponse` empty is what you want almost every time. If you assign markup to it, that markup replaces the target instead.




Demo 03 after clicking both buttons. Only the panel and the stat card were redrawn.

## Reading what the user typed

While `OnExecute` runs, `aAction.Params` holds the values the browser sent: the fields of a submitted form, each under the `Name` of its field, plus any query string values. A `dtSubmit` action follows the form's `Method`, which is `fmPost` by default. This handler from Demo 05 inserts a row from the fields `name` and `email`:

```
procedure TfrmDatabase.AddCustomerExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  FDConnection1.ExecSQL(CS_INSERT, [aAction.Params.Values['name'],
    aAction.Params.Values['email']]);
  FDQuery1.Refresh;
end;
```

> **Tip.** To make a plain button, use a `TsgcHTMLComponent_ButtonGroup` with one item, and set the item's `Text` and `ButtonStyle`. Do not reach for `TsgcHTMLComponent_PushButton`: despite the name, it is an industrial instrument control, not an ordinary button.

## Threads and the lock

The web server answers requests on its own threads, not on the main thread of your application. That has two consequences.

**Your handlers run one at a time.** The DesignServer takes a single lock around every page render and every action handler, shared by all designed pages in the application. While one handler runs, no other handler runs and no page is being rendered. You can therefore change component properties in `OnExecute` without worrying that another browser reads them halfway. Demo 05 relies on the same lock to share one FireDAC connection between all browsers.

**Your handlers must not touch the desktop window.** The lock protects the sgcHTML components, not the VCL. Setting `Label1.Caption`, showing a dialog or changing any other VCL control from `OnExecute` is a classic cross-thread bug that can freeze or crash the program. Change sgcHTML components and your own data, and leave the VCL controls alone.

## One page for every visitor

One designed form is one page, and every browser that opens its URL gets that same page. When a handler changes a component, the change is part of the page for everyone. In Demo 03 the click counter is a plain form field, so two people clicking *Add one* in two browsers increment the same number, and each of them sees the total.

Note that only the browser that clicked gets the redrawn target straight away. The others see the change when they reload the page, or when their own action redraws that component.

This is the main difference from uniGUI, where every session gets its own instances of the forms. A shared page suits dashboards, status boards, kiosks and tools for one user. When each user needs their own data, you need sessions and sign-in: `TsgcHTMLAuth` together with a session store such as `TsgcHTMLSessionStore_Memory`, applied through the `Auth` property of the HTMX engine. That is the code-first path, beyond the QuickStart. The help topic `HTML_Sessions_Auth.htm` describes it.

## Encode what users type

`Body` and many other text properties are raw HTML. Whatever you assign to them goes into the page as markup. That is what lets you write `<p>` or `<strong>` in a panel body, and it is also why you must never put user input there directly. A visitor who types a `<script>` tag into a form field would otherwise run script in every browser that shows the page.

The rule is simple: pass every value that came from the browser through `sgcHTMLEncode`, in the unit `sgcHTMX_Helpers`, before you put it inside HTML. Demo 04 does this for each field it echoes back.

The same applies to SQL. Pass user values as parameters, as Demo 05 does with `:name` and `:email`, and never paste them into the SQL text.

## Data binding

Every sgcHTML component has a `DataSource` and a `DataAutoRefresh` property. Bind a `TsgcHTMLComponent_Grid` to a `TDataSource` exactly as you would bind a `TDBGrid`, set `DataAutoRefresh` to `True`, and the grid reloads its rows by itself whenever the dataset changes. In Demo 05 the handlers run their SQL and call `FDQuery1.Refresh`. The grid follows, and because it is the action's `Target`, the new rows appear in the browser.

Two things to know:

- `DataAutoRefresh` works by taking over the `OnDataChange` event of the `TDataSource`. Use one auto-refreshing component per data source, and do not assign your own `OnDataChange` handler to that data source.
- To fill a component once, without binding, call its `LoadFromDataSet` method. About two dozen components have one, among them the Grid, the DataTable, the Chart, the Form and the Select. The parameters differ by component: the Chart, for example, also asks which field holds the labels.

## Everything is inside the executable

The page links Bootstrap, htmx and, when it has a chart, Chart.js. The DesignServer serves those files from resources compiled into your executable, with a one-day cache header. There is nothing to deploy next to the exe and no CDN to reach, so the application works the same on a laptop, on a server without internet access and inside a closed network.

The DesignServer only answers its own page, the script and style files it carries, and its action requests. Any other path that nobody else handles gets a `404 Not Found`. If your application already has an `OnCommandGet` handler on the HTTP server, that handler runs first, and the DesignServer only answers requests your handler left unanswered.

## Summary: who does what on a click

| Step | Who | What happens |
|---|---|---|
| 1 | Browser (htmx) | The user clicks the `Source` component. htmx sends a request to the action's route. |
| 2 | `TsgcHTTPServer` | Receives the request on a server thread. |
| 3 | `TsgcHTMLDesignServer` | Recognizes the action route, takes the lock and fills `aAction.Params`. |
| 4 | Your `OnExecute` | Changes component properties, runs SQL, updates your data. |
| 5 | `TsgcHTMLDesignSurface` | Renders the `Target` again, unless you filled `aResponse`. |
| 6 | Browser (htmx) | Replaces the old target with the new HTML. The rest of the page stays as it was. |

# 4. The QuickStart demos

The quickest way to learn sgcHTML is to run a small program, look at its form and change a property. That is what the six QuickStart demos are for. Each one adds a single idea to the one before it: a page, a layout, a click, a form, a database and finally a site with several pages. You find them in the `Demos\60.HTML\00.QuickStart` folder of sgcWebSockets, one subfolder per demo, each with a `readme.html` written by the author of the demo.

Every demo runs the same way. Open its `.dpr` file in Delphi with **File, Open Project** and press **F9** to run it. A small desktop window appears with the address of the page and an **Open in browser** button. Click the button, or type the address into any browser on the same machine, and the page opens. Close the desktop window to stop the web server.

Each demo listens on its own port, from 8081 to 8086, so you can run several of them at the same time and keep their pages open side by side. Two programs cannot listen on the same port at once, so close a demo before you run the same one again.

| Demo | Project | Address | What it teaches |
|---|---|---|---|
| 01.HelloWorld | `HelloWorld.dpr` | `http://127.0.0.1:8081/` | One form becomes one web page, with no code |
| 02.Components | `Components.dpr` | `http://127.0.0.1:8082/` | The main components and how the form layout becomes rows and columns |
| 03.Events | `Events.dpr` | `http://127.0.0.1:8083/` | A click in the browser runs a Delphi event handler |
| 04.Forms | `FormsDemo.dpr` | `http://127.0.0.1:8084/` | A web form posts what the user typed, and Delphi code reads it |
| 05.Database | `Database.dpr` | `http://127.0.0.1:8085/` | A grid bound to FireDAC, with add, edit and delete |
| 06.Pages | `Pages.dpr` | `http://127.0.0.1:8086/` | Several forms, one web site, one shared menu |

> **Note.** The QuickStart demos need sgcWebSockets 2026.10 or later, All-Access (which includes sgcHTML), the sgcHTML package or the sgcHTML trial, with the packages installed so the sgcHTML components appear in the Tool Palette. Delphi 10.4 or later is recommended, so the IDE shows the Web Design tab for the form. Demo 05 also uses FireDAC with the SQLite driver, which ships with Delphi. No internet connection is needed: Bootstrap, htmx and Chart.js are embedded in the executable.

> **Tip.** To see every property of a form at once, right click the form in the designer and choose **View as Text**. The DFM excerpts in this chapter come from that view. Choose **View as Form** to go back to the designer.

## 4.1 Demo 01: Hello World

Folder `01.HelloWorld`, project `HelloWorld.dpr`, port 8081.

### What you will build

The smallest sgcHTML web application there is. Four components on a Delphi form become a styled web page with one Bootstrap card that says Hello World. You write no HTML, no routes and no request handler. The only code in the unit starts the server, stops it and opens the browser.

Figure 4.1. The Hello World page at http://127.0.0.1:8081/. The Surface Title, Hello sgcHTML, becomes the page title shown in the browser tab.

The desktop window of the demo is not the page. It shows a placeholder box where the panel sits, the address label and the Open in browser button. The page itself only exists in the browser.

## Components on the form

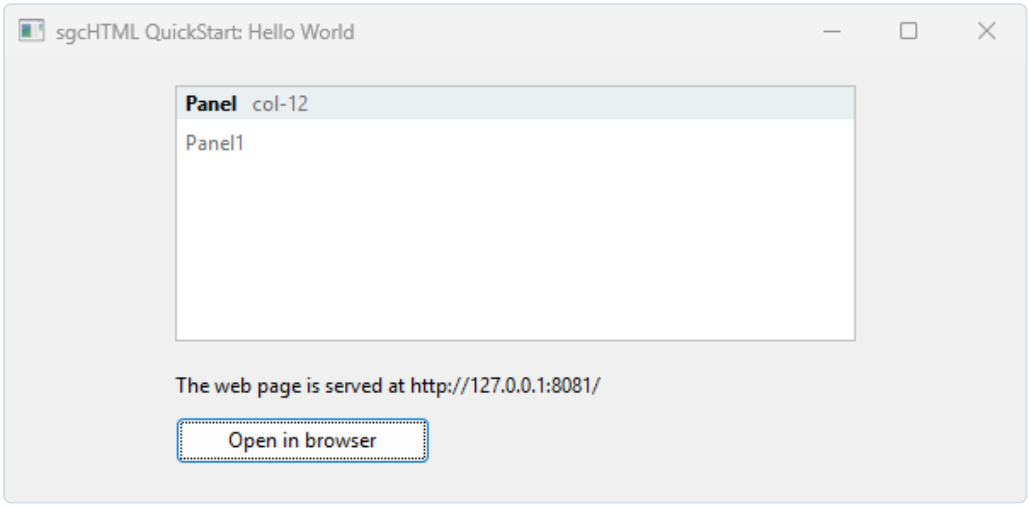



Figure 4.2. The form in Delphi: the box is the sgcHTML panel, the label and button below it are plain VCL controls that do not appear on the web page.

| Component | Class | Properties you set |
|---|---|---|
| `HTTPServer` | `TsgcHTTPServer` | `Port = 8081` |
| `DesignServer` | `TsgcHTMLDesignServer` | `Server = HTTPServer`, `Surface = Surface`, `PagePath = '/'` |
| `Surface` | `TsgcHTMLDesignSurface` | `Title = 'Hello sgcHTML'` |
| `Panel1` | `TsgcHTMLComponent_Panel` | `Title = 'Hello World'`, `Body` = the welcome sentence. `Left = 100` and `Width = 400` on a form 600 pixels wide. |
| `lblURL` | `TLabel` | `Caption` = the address of the page. Desktop window only. |
| `btnOpen` | `TButton` | `Caption = 'Open in browser'`, `OnClick = btnOpenClick`. Desktop window only. |

The first three components are the plumbing that every sgcHTML page needs, and you will meet them again in every demo:

- **HTTPServer** is the web server. It listens on a port and answers the browser.
- **DesignServer** is the glue. It connects the web server to the page. When a browser asks for its `PagePath`, here `'/'`, it answers with the page built from this form, plus the Bootstrap files the page needs.

- **Surface** is the page itself: its `Title`, which the browser shows in the tab, and every sgcHTML component dropped on the form.

The fourth component, `Panel1`, is the only thing on the page. A panel is rendered as a Bootstrap card with a header (`Title`) and a body (`Body`). This is how the four components look in `uHelloWorld.dfm`:

```
object Panel1: TsgcHTMLComponent_Panel
  Left = 100
  Top = 16
  Width = 400
  Height = 150
  Title = 'Hello World'
  Body =
    'Welcome to sgcHTML. This page was designed on a Delphi form and is' +
    ' served to your browser without a single line of HTML.'
end
```

```
object HTTPServer: TsgcHTTPServer
  Port = 8081
  Left = 536
  Top = 16
end
object DesignServer: TsgcHTMLDesignServer
  Surface = Surface
  Server = HTTPServer
  PagePath = '/'
  Left = 536
  Top = 72
end
object Surface: TsgcHTMLDesignSurface
  Title = 'Hello sgcHTML'
  Left = 536
  Top = 128
end
```

Look at the position of the panel. The form is 600 pixels wide and a Bootstrap row has 12 columns, so the 400 pixel panel gets 8 columns, and its left offset of 100 pixels centers it. Where a component sits on the form is where it sits on the page.

### The code

The `uses` clause lists one unit per sgcHTML component. The IDE adds them for you when you drop the components on the form. `ShellAPI` is there for the `ShellExecute` call that opens the browser.

```
uses
  Windows, Classes, Controls, Forms, StdCtrls, ShellAPI,
  // sgc
  sgcHTTP_Server, sgcHTMLDesign_Server_Indy, sgcHTMLDesign_Surface,
  sgcHTML_Component_Panel;
```

And this is every line of code in the demo:

```
procedure TfrmHelloWorld.FormCreate(Sender: TObject);
begin
  { Start listening. From now on http://127.0.0.1:8081/ shows the page. }
  HTTPServer.Active := True;
end;


procedure TfrmHelloWorld.FormDestroy(Sender: TObject);
begin
  { Stop listening before the form and its components are freed. }
  HTTPServer.Active := False;
end;


procedure TfrmHelloWorld.btnOpenClick(Sender: TObject);
begin
  { Opens the page in the default browser. Typing the address shown in the
    label into any browser on this machine works just as well. }
  ShellExecute(Handle, 'open', 'http://127.0.0.1:8081/', nil, nil,
    SW_SHOWNORMAL);
end;
```

Three tiny handlers: `FormCreate` starts the web server, `FormDestroy` stops it before the form and its components are freed, and `btnOpenClick` opens the page in the default browser. Building the page, serving the Bootstrap files and answering the browser is all done by the components.

### How it works

1. `HelloWorld.dpr` creates `frmHelloWorld`. While the form loads from its DFM, `DesignServer` hooks into `HTTPServer`.
2. `FormCreate` sets `HTTPServer.Active := True` and the server starts listening on port 8081.
3. You click **Open in browser**, and the browser asks for `/`.
4. `DesignServer` sees that `/` is its `PagePath`. It builds the page from the sgcHTML components on the form, here only `Panel1`, and sends it back as HTML. The VCL label and button are skipped.
5. The page asks for `/bootstrap.min.css` and `/bootstrap.bundle.min.js`. `DesignServer` serves both from resources embedded in the executable.
6. Any other address, for example `/hello`, is not a `PagePath`, so the server answers 404 Not Found.
7. You close the window. `FormDestroy` sets `HTTPServer.Active := False` and the server stops cleanly.

### Try this

- Select `Surface` and change `Title` in the Object Inspector to the name of your own application. Run again and look at the browser tab.
- Change `Panel1.Title` and `Panel1.Body`. The body accepts HTML, so try `Welcome to <strong>my</strong> first page`.
- Drag the panel to the left edge of the form and make it 600 pixels wide, as wide as the form. Run and reload the page: the card now fills the whole row.
- Change `HTTPServer.Port` to 8090. The address in `btnOpenClick` and in the label still says 8081, so update both, or type `http://127.0.0.1:8090/` into the browser yourself.

## 4.2 Demo 02: Components

Folder `02.Components`, project `Components.dpr`, port 8082.

### What you will build

A small dashboard made of the main sgcHTML components: a navigation bar, three stat cards, a bar chart with a card beside it, a progress bar, a text box, a drop down list and a set of tabs. Ten sgcHTML components in all, each one dropped on the form and configured in the Object Inspector. The code is the same three handlers as in Hello World, and there is still no HTML to write.

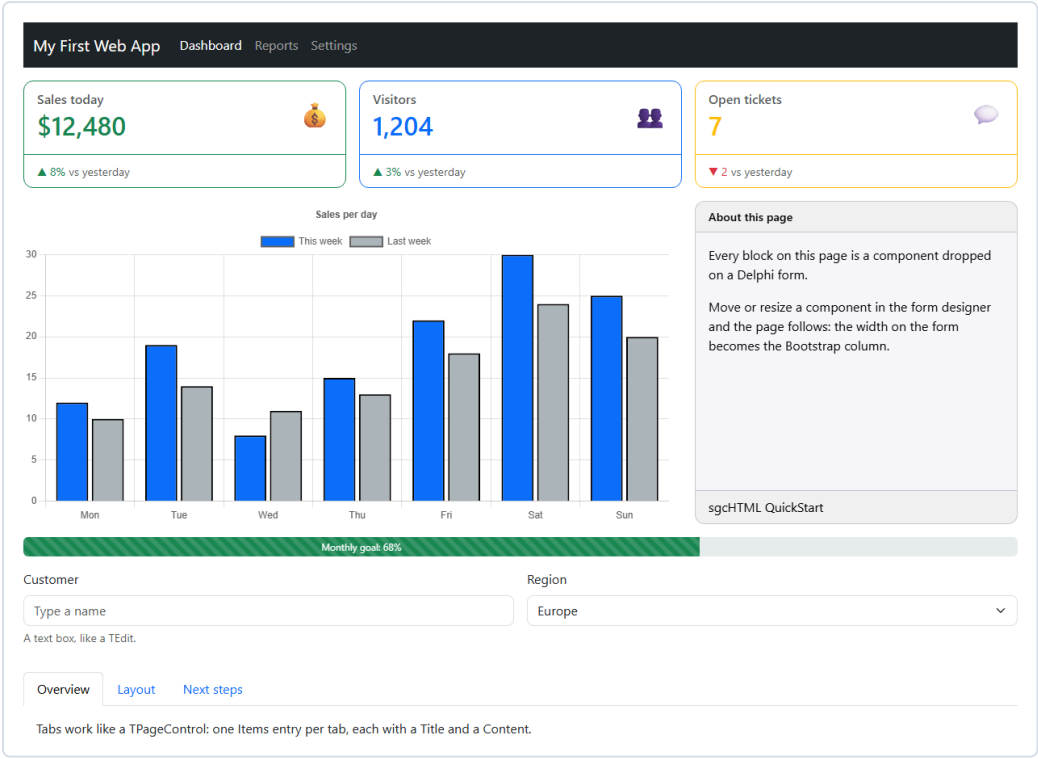



Figure 4.3. The Components dashboard. Every block is one component on the form.

### Components on the form

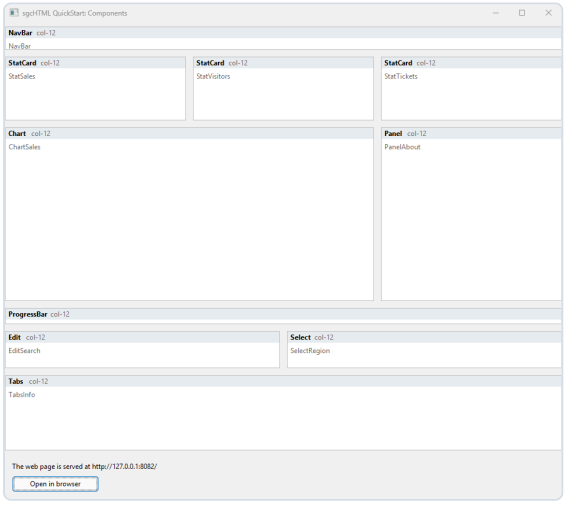



Figure 4.4. The form in Delphi: the boxes are the sgcHTML components, the label and button below them are plain VCL controls that do not appear on the web page. Compare it with the dashboard in the browser.

| Component | Class | Properties you set |
|---|---|---|
| `NavBar` | `TsgcHTMLComponent_NavBar` | `Brand = 'My First Web App'`, `BrandHref = '/'`. `Items`: Dashboard (`Href = '/'`, `Active = True`), Reports and Settings (`Href = '#'`). |
| `StatSales` | `TsgcHTMLComponent_StatCard` | `Title = 'Sales today'`, `Value = '$12,480'`, `Icon = '💰'`, `Trend = stUp`, `TrendValue = '8%'`, `Color = scSuccess`, `FooterText = 'vs yesterday'` |
| `StatVisitors` | `TsgcHTMLComponent_StatCard` | `Title = 'Visitors'`, `Value = '1,204'`, `Icon = '👥'`, `Trend = stUp`, `TrendValue = '3%'`, `FooterText`. `Color` keeps its default. |
| `StatTickets` | `TsgcHTMLComponent_StatCard` | `Title = 'Open tickets'`, `Value = '7'`, `Icon = '💬'`, `Trend = stDown`, `TrendValue = '2'`, `Color = scWarning`, `FooterText` |
| `ChartSales` | `TsgcHTMLComponent_Chart` | `ChartType = ctBar`, `Title = 'Sales per day'`, `Labels` = the seven days, `Datasets`: This week and Last week |
| `PanelAbout` | `TsgcHTMLComponent_Panel` | `Title = 'About this page'`, `Body` (two HTML paragraphs), `Footer = 'sgcHTML QuickStart'`, `CSSClass = 'h-100'` |
| `ProgressGoal` | `TsgcHTMLComponent_ProgressBar` | `Value = 68`, `ColorStyle = hcSuccess`, `Striped`, `Animated` and `ShowLabel = True`, `LabelFormat = 'Monthly goal: %d%%'`, `CSSHeight = '24px'` |
| `EditSearch` | `TsgcHTMLComponent_Edit` | `ElementName = 'search'`, `Label_ = 'Customer'`, `Placeholder = 'Type a name'`, `HelpText` |
| `SelectRegion` | `TsgcHTMLComponent_Select` | `ElementName = 'region'`, `Label_ = 'Region'`. `Options`: Europe (`Selected = True`), America, Asia. |
| `TabsInfo` | `TsgcHTMLComponent_Tabs` | `Items`: Overview (`Active = True`), Layout, Next steps, each with a `Title` and an HTML `Content` |
| `HTTPServer` | `TsgcHTTPServer` | `Port = 8082` |
| `DesignServer` | `TsgcHTMLDesignServer` | `Server = HTTPServer`, `Surface = Surface`, `PagePath = '/'` |
| `Surface` | `TsgcHTMLDesignSurface` | `Title = 'sgcHTML QuickStart: Components'` |

If you know the VCL, most of these will feel familiar. The edit box is a `TEdit`, the select is a `TComboBox` and the tabs are a `TPageControl`. A few properties need a word of explanation:

- **Collections.** `Items`, `Datasets` and `Options` are collection properties. Click the ellipsis button next to the property in the Object Inspector to open the collection editor, then add one item per menu link, chart series, list entry or tab.
- **Icon** takes an HTML character code, so `💰` is the money bag emoji you see on the Sales card.
- **Labels** lists the chart categories, each one written in double quotes and separated by commas. In every `Datasets` item, `Data` holds the numbers, also separated by commas. Chart.js draws the chart in the browser.
- **LabelFormat** works like Delphi's `Format`: `%d` is the value and `%%` is a literal percent sign. `Max` keeps its default of 100.
- **Label_** ends with an underscore because `label` is a reserved word in Pascal.
- **ElementName** is the name of the field that would be sent to the server. In this demo the text box and the list send nothing yet. Demos 03 and 04 show how the browser talks back.
- **CSSClass** adds Bootstrap classes to a component. `h-100` makes the About card as tall as the chart beside it.

Here are two of the components as they appear in `uComponents.dfm`. Notice how each collection item is written between `item` and `end`:

```
object StatSales: TsgcHTMLComponent_StatCard
  Left = 0
  Top = 60
  Width = 312
  Height = 110
  Title = 'Sales today'
  Value = '$12,480'
  Icon = '&#128176;'
  Trend = stUp
  TrendValue = '8%'
  Color = scSuccess
  FooterText = 'vs yesterday'
end
```

```
object ChartSales: TsgcHTMLComponent_Chart
  Left = 0
  Top = 182
  Width = 636
  Height = 300
  ChartType = ctBar
  Labels = '"Mon","Tue","Wed","Thu","Fri","Sat","Sun"'
  Datasets = <
    item
      Label_ = 'This week'
      Data = '12,19,8,15,22,30,25'
      BackgroundColor = '#0d6efd'
    end
    item
      Label_ = 'Last week'
      Data = '10,14,11,13,18,24,20'
      BackgroundColor = '#adb5bd'
    end>
  Title = 'Sales per day'
end
```

### How the form becomes the page

You lay out the page the way you lay out a VCL form. Three rules decide where every component lands:

- Components whose tops line up on the form share one row of the page.
- Left to right on the form is left to right on the page.
- The width on the form becomes the Bootstrap column width. The form is 960 pixels wide and a row has 12 columns, so 80 pixels make one column.

The three stat cards are 312 pixels wide and get 4 columns each. The chart is 636 pixels wide and gets 8 columns, and the 312 pixel About card next to it gets the remaining 4. The progress bar and the tabs are 960 pixels wide, so each fills a whole row, and the text box and the list share a row with 6 columns each.

### The code

The `uses` clause grows with one unit per kind of component on the form:

```
uses
  Windows, Classes, Controls, Forms, StdCtrls, ShellAPI,
  // sgc
  sgcHTTP_Server, sgcHTMLDesign_Server_Indy, sgcHTMLDesign_Surface,
  sgcHTML_Component_NavBar, sgcHTML_Component_StatCard,
  sgcHTML_Component_Chart, sgcHTML_Component_Panel,
  sgcHTML_Component_ProgressBar, sgcHTML_Component_Edit,
  sgcHTML_Component_Select, sgcHTML_Component_Tabs;
```

`FormCreate`, `FormDestroy` and `btnOpenClick` are the same as in Demo 01, except for the port: this demo listens on `8082`. The demo adds no other code.

### How it works

1. `FormCreate` sets `HTTPServer.Active := True`, the only line needed to start the web application.
2. The browser asks for `/`. `DesignServer` reads the sgcHTML components on the form and turns their positions and widths into Bootstrap rows and columns. The VCL label and button are skipped.
3. The page loads `/bootstrap.min.css` and `/bootstrap.bundle.min.js` and, because it holds a chart, `/chart.umd.min.js`. All three are embedded in the application and served by `DesignServer`.
4. Chart.js draws the bars in the browser from `Labels` and `Datasets`.
5. Clicking a tab switches its content in the browser. No request is sent to the server.

### Try this

- Change `StatSales.Value` to `'$15,000'` and its `Trend` to `stDown`. Run and reload the page.
- Set `ChartSales.ChartType` to `ctLine` and see the same data as a line chart.
- Open `ChartSales.Datasets`, add a third item with `Label_ = 'Two weeks ago'`, seven numbers in `Data` and a `BackgroundColor` such as `'#198754'`.
- Add a fourth entry to `SelectRegion.Options` with `Value = 'af'` and `Text = 'Africa'`, or a fourth tab to `TabsInfo.Items`.

## 4.3 Demo 03: Events

Folder `03.Events`, project `Events.dpr`, port 8083.

### What you will build

If you come from the VCL or uniGUI, this demo is your `Button1Click`. The page has two buttons. **Say hello** writes a greeting with the server time into a panel, and **Add one** increments a click counter shown in a stat card. A click in the browser runs a Delphi event handler on the server, the handler changes a property of a component, and only that component is sent back and swapped into the page. No page reload, no JavaScript and no request code.

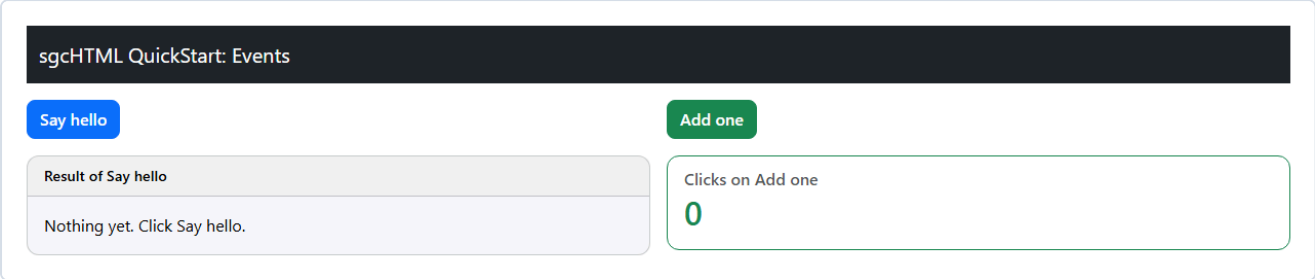



Figure 4.5. The Events page when it opens. The panel says Nothing yet and the counter shows 0.

### Components on the form

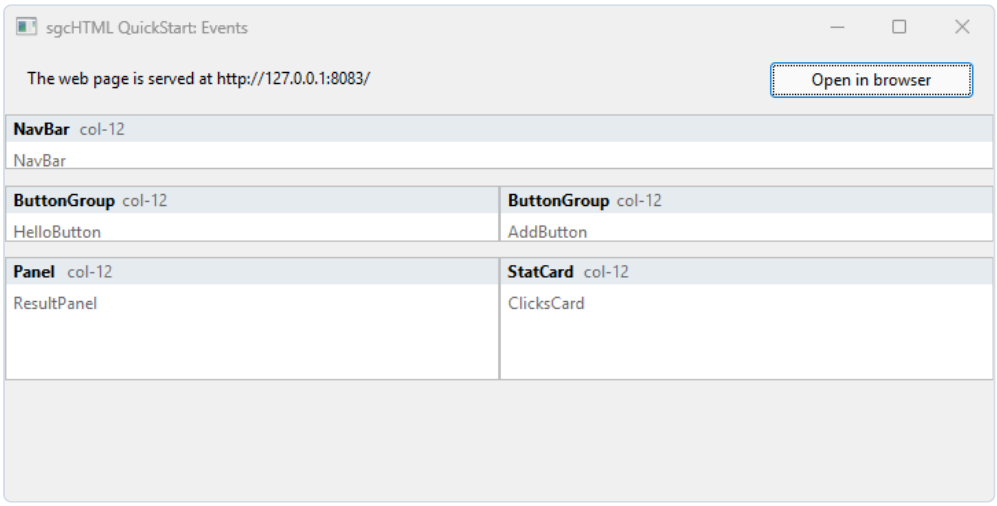



Figure 4.6. The form in Delphi: the boxes are the sgcHTML components, the label and button above them are plain VCL controls that do not appear on the web page.

| Component | Class | Properties you set |
|---|---|---|
| `NavBar` | `TsgcHTMLComponent_NavBar` | `Brand = 'sgcHTML QuickStart: Events'`, `CSSClass = 'mb-3'` |
| `HelloButton` | `TsgcHTMLComponent_ButtonGroup` | `Items`: one item, `Text = 'Say hello'`. `AriaLabel = 'Say hello'`. |
| `AddButton` | `TsgcHTMLComponent_ButtonGroup` | `Items`: one item, `Text = 'Add one'`, `ButtonStyle = bsSuccess`. `AriaLabel = 'Add one'`. |
| `ResultPanel` | `TsgcHTMLComponent_Panel` | `Title = 'Result of Say hello'`, `Body = 'Nothing yet. Click Say hello.'` |

| Component | Class | Properties you set |
|---|---|---|
| `ClicksCard` | `TsgcHTMLComponent_StatCard` | `Title = 'Clicks on Add one'`, `Value = '0'`, `Color = scSuccess` |
| `HTTPServer` | `TsgcHTTPServer` | `Port = 8083` |
| `DesignServer` | `TsgcHTMLDesignServer` | `Server = HTTPServer`, `Surface = Surface`, `PagePath = '/'` |
| `Surface` | `TsgcHTMLDesignSurface` | `Title = 'sgcHTML QuickStart: Events'`, `CompactRows = True`, two `Actions` |
| `Surface.Actions[0]` | `TsgcHTMLDesignAction` | `ActionName = 'SayHello'`, `Source = HelloButton`, `Target = ResultPanel`, `OnExecute = HelloActionExecute` |
| `Surface.Actions[1]` | `TsgcHTMLDesignAction` | `ActionName = 'AddOne'`, `Source = AddButton`, `Target = ClicksCard`, `OnExecute = AddActionExecute` |

sgcHTML has no separate button component. A button is a `TsgcHTMLComponent_ButtonGroup` with one item in `Items`, and `ButtonStyle` picks its Bootstrap color. `AriaLabel` is the name screen readers announce for the group. `CompactRows = True` on the Surface drops the extra row wrapper around a component that already spans the full width, here the menu bar, so the page markup stays lean.

The new idea is the **action**. An action is an item of `Surface.Actions` with three parts:

- **Source** is the component the user clicks.
- **Target** is the component that is drawn again when the handler returns.
- **OnExecute** is the Delphi code that runs. It plays the role of `OnClick`.

`Trigger` is left at its default, `dtClick`, so a click on the Source fires the action. To create an action yourself, select `Surface`, click the ellipsis button of `Actions` and choose Add New. Pick the Source and the Target from their drop down lists, then double click `OnExecute` on the Events tab of the Object Inspector. The IDE creates the empty handler for you, exactly as it does for `Button1Click`. This is the result in `uEvents.dfm`:

```
object Surface: TsgcHTMLDesignSurface
  Title = 'sgcHTML QuickStart: Events'
  CompactRows = True
  Actions = <
    item
      ActionName = 'SayHello'
      Source = HelloButton
      Target = ResultPanel
      OnExecute = HelloActionExecute
    end
    item
      ActionName = 'AddOne'
      Source = AddButton
      Target = ClicksCard
      OnExecute = AddActionExecute
    end>
  Left = 280
  Top = 264
end
```

## The code

The form class declares the two action handlers next to the usual three, and a private field `FClicks` for the counter. It is a plain form field, exactly as in a desktop application:

```
type
  TfrmEvents = class(TForm)
    { The desktop window: the address of the page and a shortcut to open
      it. Plain VCL controls like these never appear on the web page. }
    lblURL: TLabel;
    btnOpen: TButton;
    { The web page. Every sgcHTML component on the form becomes part of it,
      in rows and columns that follow where it sits on the form. }
    NavBar: TsgcHTMLComponent_NavBar;
    HelloButton: TsgcHTMLComponent_ButtonGroup;
    AddButton: TsgcHTMLComponent_ButtonGroup;
    ResultPanel: TsgcHTMLComponent_Panel;
    ClicksCard: TsgcHTMLComponent_StatCard;
    { The plumbing: only properties, no code. }
    HTTPServer: TsgcHTTPServer;
    DesignServer: TsgcHTMLDesignServer;
    Surface: TsgcHTMLDesignSurface;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnOpenClick(Sender: TObject);
    procedure HelloActionExecute(Sender: TObject;
      aAction: TsgcHTMLDesignAction; var aResponse: string);
    procedure AddActionExecute(Sender: TObject;
      aAction: TsgcHTMLDesignAction; var aResponse: string);
  private
    { How many times "Add one" was clicked: a plain form field, exactly as
      in a desktop application. }
    FClicks: Integer;
  end;
```

`FormCreate`, `FormDestroy` and `btnOpenClick` are the same as in Demo 01, except for the port: this demo listens on `8083`.

And here are the two handlers, with the comments the demo author wrote above them. Each one changes a property of its Target and nothing else:

```
{ OnExecute is your OnClick. This one belongs to Surface.Actions[0], whose
  Source is HelloButton and whose Target is ResultPanel.

  It runs on the server when "Say hello" is clicked in the browser. Change
  the properties of your components here. When it returns, the Target
  (ResultPanel) is rendered again and swapped into the page in place,
  without a reload. Leave aResponse empty to get exactly that.

  It runs on a server thread, not on the main thread, so it must not touch
  the VCL controls of this desktop window (lblURL, btnOpen). The sgcHTML
  components are fine: the server runs one action at a time. }
procedure TfrmEvents.HelloActionExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  ResultPanel.Body := 'Hello from Delphi! The server time is ' +
    TimeToStr(Now) + '.';
end;


{ The same idea for Surface.Actions[1]: Source is AddButton, Target is
  ClicksCard. The counter is kept in a form field, so every click shows the
  next number. There is one form for the whole server, so every browser
  sees and increments the same counter. }
procedure TfrmEvents.AddActionExecute(Sender: TObject;
```

```
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  Inc(FClicks);
  ClicksCard.Value := IntToStr(FClicks);
end;
```

Notice what the handlers do not do. They do not build HTML, they do not write a response and they do not tell the browser what to update. They leave `aResponse` empty, and that is exactly what asks the server to render the Target again and send it back.

## How it works

1. When the page is built, each button that is the Source of an action is wrapped in an element that tells the browser, through htmx, to call the server on a click and where to put the answer. The page loads `/htmx.min.js` from the executable, like Bootstrap.
2. You click **Say hello**. The browser sends the request for that action to the server.
3. `DesignServer` finds the action and runs `HelloActionExecute` on a server thread. The handler sets `ResultPanel.Body`.
4. The handler returns with `aResponse` empty, so `DesignServer` renders the Target, `ResultPanel`, and sends only that piece of HTML back.
5. htmx swaps the new panel into the page in place. The rest of the page does not move and there is no reload.
6. **Add one** works the same way: `AddActionExecute` increments `FClicks`, writes it into `ClicksCard.Value`, and only the card is redrawn.
7. Reload the page and the panel and the card keep their last values, because they live in the components on the server.

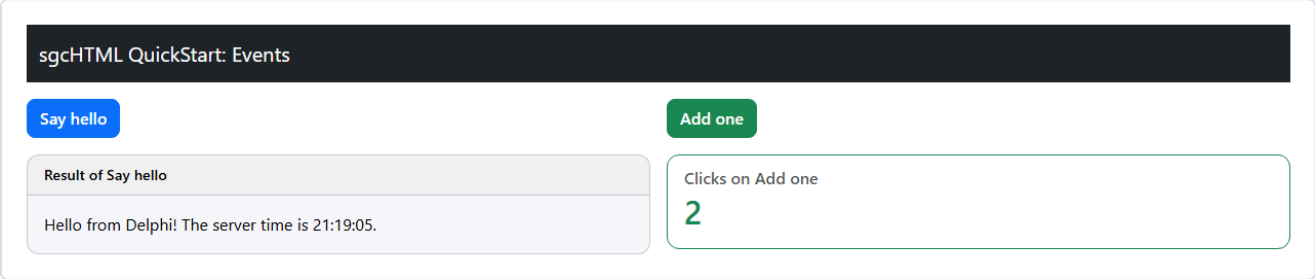



Figure 4.7. After one click on Say hello and two on Add one. Only the panel and the card changed.

**Watch out.** Action handlers run on a server thread, not on the main thread, so they must not touch the VCL controls of the desktop window, such as `lblURL` or `btnOpen`. The sgcHTML components are fine to change, because the server runs one action at a time.

**Note.** One designed form is one page, shared by every visitor. There is one `frmEvents` for the whole server, so every browser sees and increments the same counter. If you come from uniGUI, where each session gets its own copy of a form, keep this in mind: anything you store in a form field is shared by everyone who opens the page.

## Try this

- Change `HelloActionExecute` so the greeting also shows the date, with `DateToStr(Now)`.

- Add a Reset button. Make `HelloButton` and `AddButton` 240 pixels wide and drop a third `TsgcHTMLComponent_ButtonGroup` named `ResetButton` at `Left = 480` with the same `Top`, with one item `Text = 'Reset'`. Add an action with `Source = ResetButton`, `Target = ClicksCard` and a handler that sets `FClicks := 0` and `ClicksCard.Value := '0'`.
- Set `Confirm = 'Reset the counter?'` on that new action. The browser now asks before it runs.
- Open the page in two browser windows, click Add one in the first, then click Add one in the second. The counter keeps counting from the number the first window left.

## 4.4 Demo 04: Forms

Folder `04.Forms`, project `FormsDemo.dpr`, port 8084.

### What you will build

In a VCL application you read `Edit1.Text` when the user clicks a button. On the web the user types into a browser, so the values have to travel to the server first. This demo shows that sgcHTML makes it feel the same: you design the form in the Object Inspector, you add one item to `Surface.Actions`, and you write one `OnExecute` handler that reads the values by name, checks them and writes the answer into a card.

The page has two columns. On the left, `ContactForm` asks for a name, an email and a message. On the right, `ResultPanel` shows what Delphi received. When the user presses **Send**, only the card on the right is redrawn and the page never reloads.

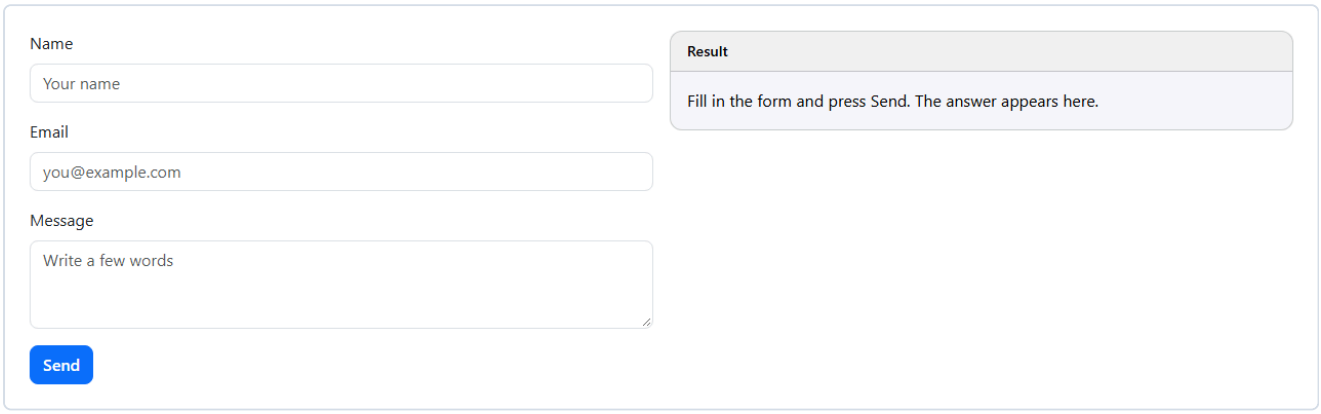



Figure 4.8. The Forms page when it opens.

### Components on the form

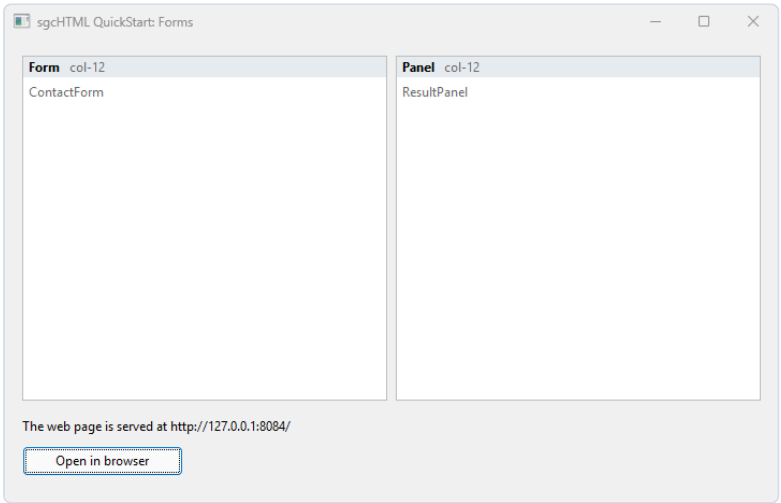



Figure 4.9. The form in Delphi: the two boxes are the sgcHTML form and the result panel, the label and button below them are plain VCL controls that do not appear on the web page.

| Component | Class | Properties you set |
|---|---|---|
| `ContactForm` | `TsgcHTMLComponent_Form` | `Fields`: name (`Label_ = 'Name'`, `Required = True`), email (`FieldType = ftEmail`, `Label_ = 'Email'`) and message (`FieldType = ftTextArea`, `Label_ = 'Message'`), each with a `Placeholder`. `SubmitText = 'Send'`. `Method` keeps its default, `fmPost`. |
| `ResultPanel` | `TsgcHTMLComponent_Panel` | `Title = 'Result'`, `Body = 'Fill in the form and press Send. The answer appears here.'` |
| `HTTPServer` | `TsgcHTTPServer` | `Port = 8084` |
| `DesignServer` | `TsgcHTMLDesignServer` | `Server = HTTPServer`, `Surface = Surface`, `PagePath = '/'` |
| `Surface` | `TsgcHTMLDesignSurface` | `Title = 'sgcHTML QuickStart: Forms'`, one `Actions` item |
| `Surface.Actions[0]` | `TsgcHTMLDesignAction` | `ActionName = 'SendContact'`, `Trigger = dtSubmit`, `Source = ContactForm`, `Target = ResultPanel`, `OnExecute = SendContactExecute` |
| `lblURL`, `btnOpen` | `TLabel`, `TButton` | Desktop window only, as in every demo |

The form and the panel have the same `Top`, so they share one row of the page. The form is 720 pixels wide, so one of the 12 Bootstrap columns is 60 pixels, and each 340 pixel component gets 6 columns, half of the row.

The inputs of a web form are items of the `Fields` collection, which you edit much like the columns of a grid. `FieldType` picks the kind of input: a plain text box when you leave it alone, `ftEmail` for an email address, `ftTextArea` for a box with several lines. This is `ContactForm` in `uForms.dfm`:

```
object ContactForm: TsgcHTMLComponent_Form
  Left = 16
  Top = 16
  Width = 340
  Height = 320
  Fields = <
    item
      Name = 'name'
      Label_ = 'Name'
      Placeholder = 'Your name'
      Required = True
    end
    item
      FieldType = ftEmail
      Name = 'email'
      Label_ = 'Email'
      Placeholder = 'you@example.com'
    end
    item
      FieldType = ftTextArea
      Name = 'message'
      Label_ = 'Message'
      Placeholder = 'Write a few words'
    end>
  SubmitText = 'Send'
end
```

> **Tip.** The `Name` of a field is its key. The browser posts every input as name=value, using the `Name` you gave the item in `ContactForm.Fields`. That is why the handler reads `Params.Values['name']`, `Params.Values['email']` and `Params.Values['message']`.

The action links the form to the handler and to the card it redraws. `Trigger = dtSubmit` fires it when the form is submitted, and the submit follows the form's `Method`, a POST here:

```
  object Surface: TsgcHTMLDesignSurface
    Title = 'sgcHTML QuickStart: Forms'
    Actions = <
      item
        ActionName = 'SendContact'
        Trigger = dtSubmit
        Source = ContactForm
        Target = ResultPanel
        OnExecute = SendContactExecute
      end>
    Left = 656
    Top = 364
  end
```

## The code

The `interface` section lists the units of the components on the form, as usual:

```
uses
  Classes, Controls, Forms, StdCtrls,
  // sgc
  sgcHTTP_Server, sgcHTMLDesign_Server_Indy, sgcHTMLDesign_Surface,
  sgcHTMLDesign_Actions, sgcHTML_Component_Form, sgcHTML_Component_Panel;
```

The `implementation` section adds two more. `sgcHTML_Enums` declares the colors `hcWarning` and `hcSuccess`, and `sgcHTMX_Helpers` declares `sgcHTMLEncode`. They are not units of a component on the form, so you add them to the `uses` clause yourself when your code needs them:

```
uses
  Windows, SysUtils, ShellAPI,
  // sgc
  sgcHTML_Enums, sgcHTMX_Helpers;
```

`FormCreate`, `FormDestroy` and `btnOpenClick` are the same as in Demo 01, except for the port: this demo listens on `8084`.

And this is the handler, with the comments of the demo author:

```
{ Runs when the browser submits ContactForm. Double click OnExecute of the
  item in Surface.Actions to create a handler like this one.

  aAction.Params holds the posted values. The key of each value is the
  Name of its item in ContactForm.Fields, so the field named 'name' is
  read with Params.Values['name'].

  aResponse stays empty, so the server draws ResultPanel again and the
  browser swaps only that card. The page does not reload and the form
  keeps what the user typed.
```

```
  This code runs in a thread of the web server, not in the main thread.
  Change sgcHTML components here, never VCL controls such as lblURL. }
procedure TfrmForms.SendContactExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
var
  vName, vEmail, vMessage: string;
begin
  vName := Trim(aAction.Params.Values['name']);
  vEmail := Trim(aAction.Params.Values['email']);
  vMessage := Trim(aAction.Params.Values['message']);

  { The browser already refuses an empty name, because the field is
    Required. A request can come from anywhere, so check it here too. }
  if vName = '' then
  begin
    ResultPanel.Color := hcWarning;
    ResultPanel.Body := 'Please type your name.';
    Exit;
  end;

  { Body is raw HTML. Pass everything the user typed through sgcHTMLEncode
    first: '<b>' is then shown as text and never runs as markup, which is
    what keeps scripts typed into the form out of the page. }
  ResultPanel.Color := hcSuccess;
  ResultPanel.Body :=
    '<p>Thank you, <strong>' + sgcHTMLEncode(vName) + '</strong>. ' +
    'Delphi received:</p>' +
    '<ul class="mb-0">' +
    '<li>Email: ' + sgcHTMLEncode(vEmail) + '</li>' +
    '<li>Message: ' + sgcHTMLEncode(vMessage) + '</li>' +
    '</ul>';
end;
```

Read it in three steps. First it takes the three values out of `aAction.Params` and trims them. Then it checks the name: when it is empty, the card turns yellow with a short message and the handler exits. Otherwise the card turns green and its `Body` lists what the user typed. The panel's `Color` property sets the color of the whole card, and every value goes through `sgcHTMLEncode` before it becomes part of the HTML.

## How it works

1. `FormCreate` starts the server on port 8084, and the browser asks for `/`.
2. `DesignServer` renders the page. Because of the action, the form tag carries an htmx `hx-post` with a route such as `/sgcaction/n2`, and an `hx-target` pointing at the result card.
3. You type a name, an email and a message and press **Send**. Because the name field is `Required`, the browser refuses to send the form while the name is empty.
4. htmx posts the fields to the action route instead of loading a new page.
5. `DesignServer` fills `aAction.Params` with the posted values, keyed by the `Name` of each field, and calls `SendContactExecute` on a server thread.
6. The handler checks the name, then writes the encoded values into `ResultPanel.Body` and sets `ResultPanel.Color`.
7. `aResponse` stays empty, so the server renders `ResultPanel` again and htmx swaps only that card into the page.

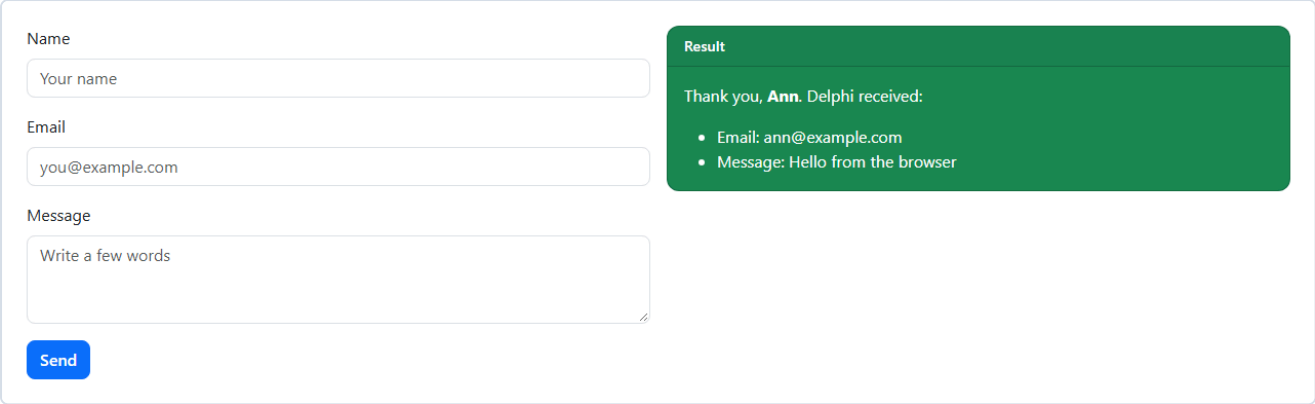



Figure 4.10. After sending the name Ann, the email ann@example.com and the message Hello from the browser. Only the Result card changed.

**Watch out.** Always encode user input. `Body` is raw HTML and is sent to the browser exactly as you write it. Text typed by a user must go through `sgcHTMLEncode` before it is added to `Body`. Otherwise a visitor could type a `<script>` tag into the form and it would run in the page.

**Note.** Check on the server too. `Required = True` makes the browser refuse an empty name, but anyone can post to the route without a browser, which is why the handler checks the name again. And as in Demo 03, there is one `ResultPanel` for the whole server, so a visitor who reloads the page sees the last answer, even one another visitor sent.

## Try this

- Type `<b>bold</b>` as the name and press Send. The card shows the tags as text, because `sgcHTMLEncode` turned them into `<b>`.
- Add a phone field. Add an item to `ContactForm.Fields` with `Name = 'phone'` and `Label_ = 'Phone'`, read it in the handler with `Trim(aAction.Params.Values['phone'])` and add one more `<li>` line to `Body`, encoded like the others.
- Make the email required: set `Required = True` on the email item, then add a server check for an empty email that turns the card yellow, just like the check for the name.
- Change `SubmitText` to `'Send message'` and set `SubmitStyle` to `bsSuccess` for a green button.

## 4.5 Demo 05: Database

Folder `05.Database`, project `Database.dpr`, port 8085.

### What you will build

The classic first database application, served as a web page. A grid lists customers, and three small forms add, edit and delete them. If you have ever dropped a `TFDConnection`, a `TFDQuery`, a `TDataSource` and a `TDBGrid` on a VCL form, you already know most of this demo. The only change is the grid: `TsgcHTMLComponent_Grid` is bound to the `TDataSource` exactly like a `TDBGrid`, but it is drawn in the browser.

The data lives in an in-memory SQLite database that is created and filled with four customers when the application starts. Nothing is written to disk, and every run starts again with the same four rows.

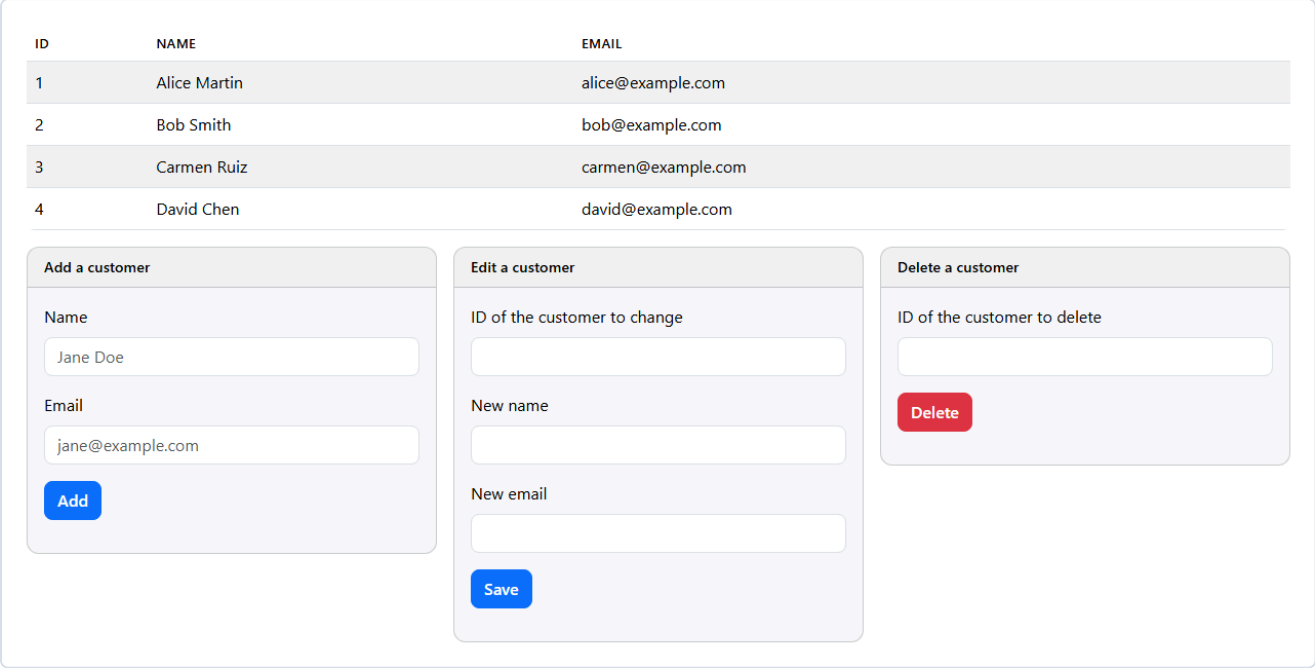



Figure 4.11. The Customers page with the four starting rows.

### Components on the form

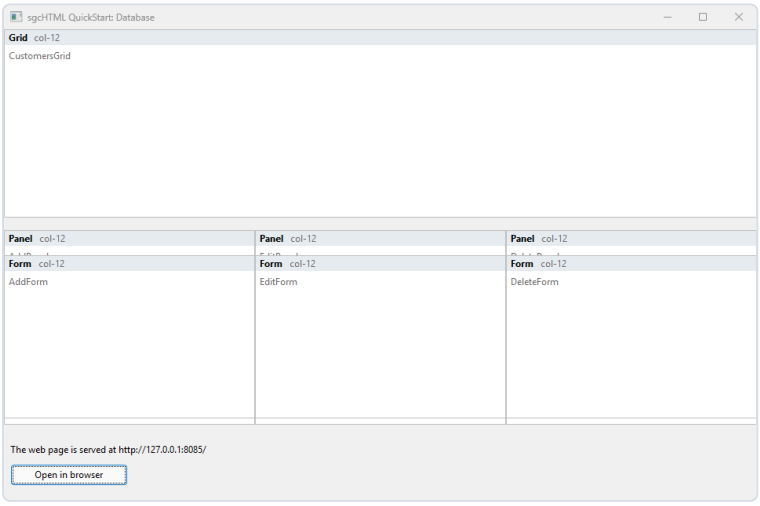



Figure 4.12. The form in Delphi: the boxes are the sgcHTML grid and three panels, each holding a form. The label and button below them are plain VCL controls that do not appear on the web page.

| Component | Class | Properties you set |
|---|---|---|
| `FDConnection1` | `TFDConnection` | `Params`: `DriverID=SQLite`, `Database=:memory:`. `LoginPrompt = False`, `ResourceOptions.SilentMode = True`. |
| `FDQuery1` | `TFDQuery` | `Connection = FDConnection1`, `SQL = 'select id, name, email from customers order by id'` |
| `DataSource1` | `TDataSource` | `DataSet = FDQuery1` |
| `CustomersGrid` | `TsgcHTMLComponent_Grid` | `DataSource = DataSource1`, `DataAutoRefresh = True`. `Columns`: id, name and email with the titles ID, Name and Email. `EmptyText = 'No customers yet'`. |
| `AddPanel`, `EditPanel`, `DeletePanel` | `TsgcHTMLComponent_Panel` | `Title` = the card header. Each is 320 pixels wide, a third of the form, and holds one form. |
| `AddForm` | `TsgcHTMLComponent_Form` | `Fields`: name (required) and email (`ftEmail`). `SubmitText = 'Add'`. |
| `EditForm` | `TsgcHTMLComponent_Form` | `Fields`: id (`ftNumber`, required), name (required) and email (`ftEmail`). `SubmitText = 'Save'`. |
| `DeleteForm` | `TsgcHTMLComponent_Form` | `Fields`: id (`ftNumber`, required). `SubmitText = 'Delete'`, `SubmitStyle = bsDanger`. |
| `Surface` | `TsgcHTMLDesignSurface` | `Title = 'Customers'`, `CompactRows = True`. Three `Actions`, all with `Trigger = dtSubmit` and `Target = CustomersGrid`. |
| `HTTPServer` | `TsgcHTTPServer` | `Port = 8085` |
| `DesignServer` | `TsgcHTMLDesignServer` | `Server = HTTPServer`, `Surface = Surface`, `PagePath = '/'` |

The grid is bound like any data-aware control. `DataAutoRefresh = True` makes it reload its rows by itself whenever `FDQuery1` changes, so no code ever fills it. Each item of `Columns` names a field of the query and gives it a title:

```
object CustomersGrid: TsgcHTMLComponent_Grid
  Left = 0
  Top = 0
  Width = 960
  Height = 240
  DataSource = DataSource1
  DataAutoRefresh = True
  Columns = <
    item
      Name = 'id'
      Title = 'ID'
    end
    item
      Name = 'name'
      Title = 'Name'
    end
    item
      Name = 'email'
      Title = 'Email'
    end>
  EmptyText = 'No customers yet'
end
```

Each form sits inside a panel, the same way you would drop a `TEdit` onto a `TPanel`. The panel gives the form a card with a title. The fields of a form are items of its `Fields` collection:

```
object AddPanel: TsgcHTMLComponent_Panel
  Left = 0
  Top = 256
  Width = 320
  Height = 248
  Title = 'Add a customer'
  object AddForm: TsgcHTMLComponent_Form
    Left = 0
    Top = 32
    Width = 320
    Height = 208
    Fields = <
      item
        Name = 'name'
        FieldID = 'add-name'
        Label_ = 'Name'
        Placeholder = 'Jane Doe'
        Required = True
      end
      item
        FieldType = ftEmail
        Name = 'email'
        FieldID = 'add-email'
        Label_ = 'Email'
        Placeholder = 'jane@example.com'
      end>
    SubmitText = 'Add'
  end
end
```

Two properties of a field are easy to confuse. `Name` is the key you read in the handler, for example `aAction.Params.Values['email']`. `FieldID` is the HTML id of the field on the page. Two forms here have a `name` field, so each one gets its own `FieldID`, `add-name` and `edit-name`, to keep the ids unique. `Required = True` marks the field as required in the page, so the browser does not submit the form while it is empty. `FieldType` picks the kind of input: `ftEmail` for an email address, `ftNumber` for a number.

The actions are the same idea as in Demo 03, with two differences. `Trigger = dtSubmit` fires the action when the form is submitted instead of when it is clicked, and the delete action has a `Confirm` text, so the browser asks before it deletes anything:

```
    Actions = <
      item
        ActionName = 'AddCustomer'
        Trigger = dtSubmit
        Source = AddForm
        Target = CustomersGrid
        OnExecute = AddCustomerExecute
      end
      item
        ActionName = 'EditCustomer'
        Trigger = dtSubmit
        Source = EditForm
        Target = CustomersGrid
        OnExecute = EditCustomerExecute
      end
      item
        ActionName = 'DeleteCustomer'
        Trigger = dtSubmit
        Confirm = 'Delete this customer?'
        Source = DeleteForm
        Target = CustomersGrid
        OnExecute = DeleteCustomerExecute
      end>
```

The three FireDAC components are configured exactly as in a desktop application, with the values in the table above.

### The code

The `uses` clause is the longest of all the demos, because of FireDAC. Read the comment: a VCL application needs `FireDAC.Comp.UI` and `FireDAC.VCLUI.Wait`, and the static SQLite wrapper links the engine into the executable, so no `sqlite3.dll` is needed:

```
uses
  Windows, SysUtils, Classes, Controls, Forms, StdCtrls, ShellAPI, DB,
  { FireDAC: the components on the form and the SQLite driver, with the
    SQLite engine linked into the exe so no sqlite3.dll is needed. A VCL
    application also needs FireDAC.Comp.UI and FireDAC.VCLUI.Wait, or
    FireDAC raises "Object factory ... TFDGUIxWaitCursor is missing". }
  FireDAC.Comp.Client, FireDAC.Stan.Def, FireDAC.Stan.Async, FireDAC.DApt,
  FireDAC.Phys.SQLite, FireDAC.Phys.SQLiteWrapper.Stat, FireDAC.Comp.UI,
  FireDAC.VCLUI.Wait,
  // sgc
  sgcHTTP_Server, sgcHTMLDesign_Server_Indy, sgcHTMLDesign_Surface,
  sgcHTMLDesign_Actions, sgcHTML_Component_Grid, sgcHTML_Component_Form,
  sgcHTML_Component_Panel;
```

The insert statement is a constant, because `FormCreate` and the Add handler both use it:

```
const
  { :name and :email are parameters. ExecSQL fills them, in this order,
    from the values passed next to the SQL. }
  CS_INSERT = 'insert into customers (name, email) values (:name, :email)';
```

`FormCreate` builds the table, adds four customers, opens the query and starts the server:

```
procedure TfrmDatabase.FormCreate(Sender: TObject);
begin
  { Create the table and add a few customers to start with. }
  FDConnection1.ExecSQL('create table customers (' +
    'id integer primary key autoincrement, ' +
    'name varchar(80) not null, email varchar(120))');
  FDConnection1.ExecSQL(CS_INSERT, ['Alice Martin', 'alice@example.com']);
  FDConnection1.ExecSQL(CS_INSERT, ['Bob Smith', 'bob@example.com']);
  FDConnection1.ExecSQL(CS_INSERT, ['Carmen Ruiz', 'carmen@example.com']);
  FDConnection1.ExecSQL(CS_INSERT, ['David Chen', 'david@example.com']);

  { Opening the query fills CustomersGrid, through DataSource1. }
  FDQuery1.Open;

  { Start listening. From now on http://127.0.0.1:8085/ shows the page. }
  HTTPServer.Active := True;
end;
```

`FormDestroy` and `btnOpenClick` are the same as in Demo 01, except for the port: this demo listens on `8085`.

And the three action handlers, each one SQL statement followed by a refresh:

```
{ The three handlers below follow the same two steps:

  1. Run one SQL statement with the values the user typed. aAction.Params
     holds the fields of the submitted form, by the Name of each field.
  2. Call FDQuery1.Refresh. DataAutoRefresh reloads CustomersGrid from the
     query, and because aResponse is left empty the DesignServer sends the
     redrawn grid (the Target of the action) back to the browser. }

procedure TfrmDatabase.AddCustomerExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  FDConnection1.ExecSQL(CS_INSERT, [aAction.Params.Values['name'],
    aAction.Params.Values['email']]);
  FDQuery1.Refresh;
end;


procedure TfrmDatabase.EditCustomerExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  FDConnection1.ExecSQL(
    'update customers set name = :name, email = :email where id = :id',
    [aAction.Params.Values['name'], aAction.Params.Values['email'],
    StrToIntDef(aAction.Params.Values['id'], 0)]);
  FDQuery1.Refresh;
end;


procedure TfrmDatabase.DeleteCustomerExecute(Sender: TObject;
  aAction: TsgcHTMLDesignAction; var aResponse: string);
begin
  FDConnection1.ExecSQL('delete from customers where id = :id',
    [StrToIntDef(aAction.Params.Values['id'], 0)]);
  FDQuery1.Refresh;
end;
```

The values typed in the browser reach SQL as parameters ( `:name` , `:email` , `:id` ), never pasted into the SQL text. A parameter is always treated as data, so a name such as `x'); drop table customers;--` is stored as a name and never runs as SQL. That attack is called SQL injection, and parameters are the protection against it.

## How it works

1. `FormCreate` creates the `customers` table, inserts four rows and opens `FDQuery1` . Opening the query fills `CustomersGrid` through `DataSource1` . Then the server starts on port 8085.
2. The browser asks for `/` and gets the grid with four customers and the three cards. Each form submits through htmx.
3. You fill in **Add a customer** and click **Add**. The browser posts the form, through htmx, to the action of `AddForm` . A `dtSubmit` action follows the form's `Method` , which keeps its default, a POST.
4. `DesignServer` puts the posted fields into `aAction.Params` , keyed by the `Name` of each field, and runs `AddCustomerExecute` on a server thread.
5. The handler runs the insert with parameters and calls `FDQuery1.Refresh` . Because `DataAutoRefresh` is on, the grid reloads its rows from the refreshed query.
6. The handler leaves `aResponse` empty, so `DesignServer` sends back the redrawn grid, the Target of the action, and the browser swaps it in place. The page never reloads.
7. Edit and Delete work the same way: type the ID shown in the grid, then click **Save** or **Delete**. For Delete, the browser first asks *Delete this customer?*.

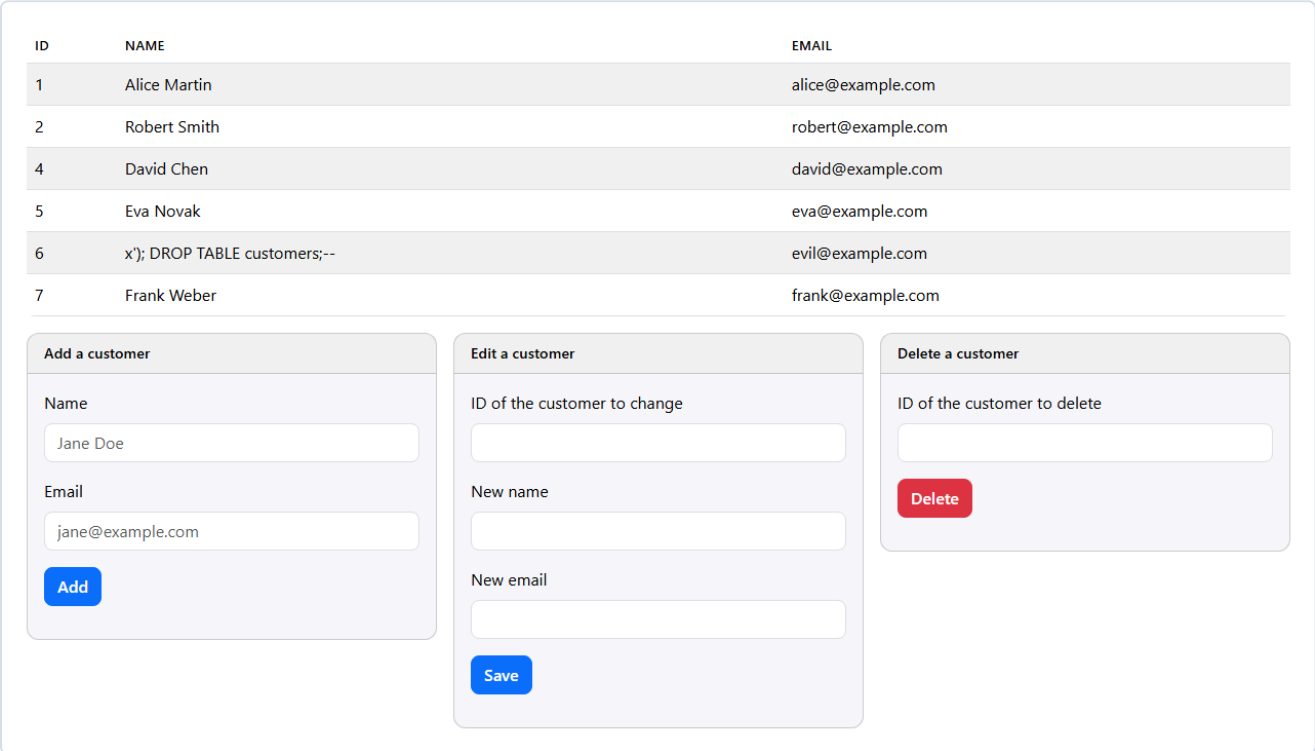



Figure 4.13. After a few changes: customer 2 renamed, customer 3 deleted and three customers added. Row 6 is an SQL injection attempt, stored as plain text.

> **Watch out.** The handlers run on a server thread, not on the main thread. `DesignServer` runs them one at a time, so two browsers never use `FDConnection1` and `FDQuery1` at the same moment. `SilentMode = True` in `ResourceOptions` stops FireDAC from trying to show an hourglass cursor from that thread. Keep both settings in mind when you move this pattern into your own application.

> **Note.** If you type an ID that does not exist into Edit or Delete, nothing changes and no message appears. The SQL statement simply matches no row, and the grid is redrawn as it was. An ID that is not a number becomes 0 through `StrToIntDef`, with the same result.

### Try this

- Type `x'); DROP TABLE customers;--` as the name of a new customer. It appears in the grid as a name, and later adds keep working.
- Set `Confirm = 'Save these changes?'` on the `EditCustomer` action, so Save asks first, just like Delete.
- Change `CustomersGrid.EmptyText`, run, and delete the four customers one by one (IDs 1 to 4) to see the new text.
- Add a phone number. Add `phone varchar(30)` to the `create table` statement and `phone` to the query's `SQL`, add a `Columns` item with `Name = 'phone'`, add a field to `AddForm` with `Name = 'phone'` and `FieldID = 'add-phone'`, then extend the insert and read `aAction.Params.Values['phone']` in `AddCustomerExecute`.

## 4.6 Demo 06: Pages

Folder `06.Pages`, project `Pages.dpr`, port 8086.

### What you will build

A small web site with three pages and a shared menu. The rule of this demo is simple: one form is one page. The main form is the Home page at `/`, `frmCustomers` is the page at `/customers` and `frmAbout` is the page at `/about`. The menu is designed once on a frame and dropped on every page form, and all three pages share one web server. The only code in the whole project starts and stops the server and opens the browser. The two page units contain no code at all.

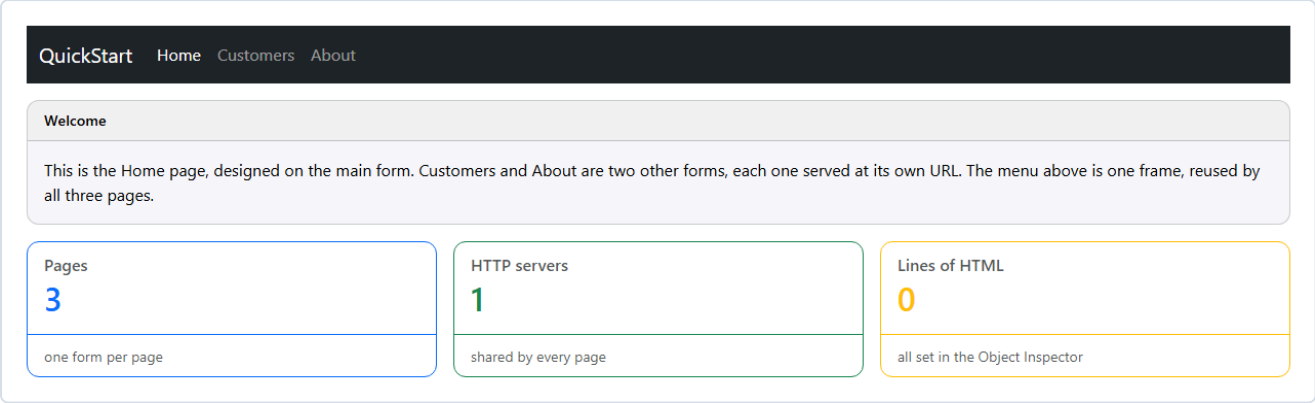



Figure 4.14. The Home page at http://127.0.0.1:8086/, with Home highlighted in the menu.

### Components on the forms

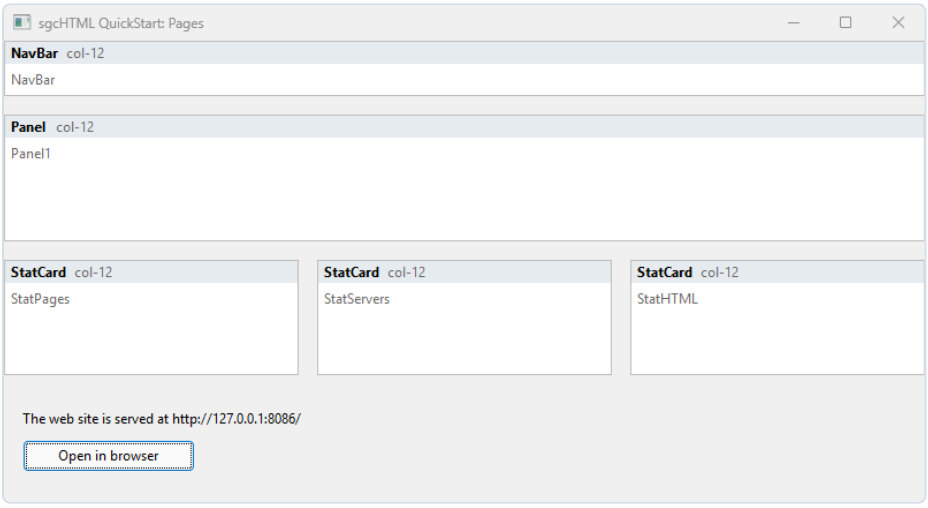



Figure 4.15. The main form in Delphi, which becomes the Home page. The NavBar box at the top is the navigation frame, the other boxes are sgcHTML components, and the label and button below them are plain VCL controls that do not appear on the web page.

This demo has four units with a form or a frame each. The table lists them unit by unit:

| Component | Class | Properties you set |
|---|---|---|
| uNavFrame: `NavBar` | `TsgcHTMLComponent_NavBar` | `Brand = 'QuickStart'`, `BrandHref = '/'`. `Items`: Home `'/'`, Customers `'/customers'`, About `'/about'`. |

| Component | Class | Properties you set |
|---|---|---|
| uMain: `fraNav` | `TfraNav` | The shared menu frame. Its copy of `NavBar.Items` has `Active = True` on Home. |
| uMain: `HTTPServer` | `TsgcHTTPServer` | `Port = 8086`. The one web server of the site. |
| uMain: `DesignServer` | `TsgcHTMLDesignServer` | `Server = HTTPServer`, `Surface = Surface`, `PagePath = '/'` |
| uMain: `Surface` | `TsgcHTMLDesignSurface` | `Title = 'Home'` |
| uMain: `Panel1` | `TsgcHTMLComponent_Panel` | `Title = 'Welcome'`, `Body` = the welcome text |
| uMain: `StatPages`, `StatServers`, `StatHTML` | `TsgcHTMLComponent_StatCard` | `Title`, `Value` (`'3'`, `'1'` and `'0'`), `Color`, `FooterText` |
| uCustomers: `DesignServer` | `TsgcHTMLDesignServer` | `Server = frmMain.HTTPServer`, `Surface = Surface`, `PagePath = '/customers'` |
| uCustomers: `Surface`, `fraNav` | `TsgcHTMLDesignSurface`, `TfraNav` | `Title = 'Customers'`. The menu with `Active = True` on Customers. |
| uCustomers: `Grid` | `TsgcHTMLComponent_Grid` | `Columns`: Name, City, Email. `Rows`: five customers, one per line, the cells separated by a Tab. |
| uAbout: `DesignServer` | `TsgcHTMLDesignServer` | `Server = frmMain.HTTPServer`, `Surface = Surface`, `PagePath = '/about'` |
| uAbout: `Surface`, `fraNav`, `Panel1` | `TsgcHTMLDesignSurface`, `TfraNav`, `TsgcHTMLComponent_Panel` | `Title = 'About'`. The menu with `Active = True` on About. A card titled About this demo. |
| uMain: `lblURL`, `btnOpen` | `TLabel`, `TButton` | Desktop window only, as in every demo |

Every page form has its own `TsgcHTMLDesignServer` and its own `TsgcHTMLDesignSurface`, and the `PagePath` of the DesignServer is the URL of that page. There is only one web server, the `TsgcHTTPServer` on the main form. The DesignServers of the other two forms point to it with `Server = frmMain.HTTPServer`. Each DesignServer answers its own URL and leaves the others alone.

The menu is a `TsgcHTMLComponent_NavBar` on a `TFrame`. This is the whole frame, `uNavFrame.dfm`. `Href` is the URL each menu entry opens:

```
object fraNav: TfraNav
  Left = 0
  Top = 0
  Width = 800
  Height = 48
  TabOrder = 0
  object NavBar: TsgcHTMLComponent_NavBar
    Left = 0
    Top = 0
    Width = 800
    Height = 48
    Brand = 'QuickStart'
    BrandHref = '/'
    Items = <
      item
        Text = 'Home'
        Href = '/'
      end
      item
        Text = 'Customers'
        Href = '/customers'
      end
      item
        Text = 'About'
        Href = '/about'
      end>
  end
end
```

Each page form drops the frame at its top and overrides the `Items` of its copy to tick `Active` on its own entry. This is the copy on the Customers page. The `inline` and `inherited` keywords are how Delphi stores a frame and the properties a form changes on it:

```
  inline fraNav: TfraNav
    Left = 0
    Top = 0
    Width = 800
    Height = 48
    TabOrder = 0
    inherited NavBar: TsgcHTMLComponent_NavBar
      Items = <
        item
          Text = 'Home'
          Href = '/'
        end
        item
          Text = 'Customers'
          Href = '/customers'
          Active = True
        end
        item
          Text = 'About'
          Href = '/about'
        end>
    end
  end
```

The Customers page shows a grid without any dataset. Its `Rows` hold the data as text, one line per row, and the DFM writes the Tab between the cells as `#9`:

```
    Rows.Strings = (
      'Alice Johnson'#9'New York'#9'alice@example.com'
      'Bob Smith'#9'London'#9'bob@example.com'
      'Carol Williams'#9'Berlin'#9'carol@example.com'
      'David Brown'#9'Paris'#9'david@example.com'
      'Emma Davis'#9'Tokyo'#9'emma@example.com')
```

And this is the DesignServer that plugs the Customers page into the main form's server:

```
  object DesignServer: TsgcHTMLDesignServer
    Surface = Surface
    Server = frmMain.HTTPServer
    PagePath = '/customers'
    Left = 664
    Top = 280
  end
```

## The code

The order in which the forms are created matters, so `Pages.dpr` deserves a look:

```
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  { The main form first: it owns the web server the other pages use. }
  Application.CreateForm(TfrmMain, frmMain);
  { The page forms. They stay hidden, they only exist to be served. }
  Application.CreateForm(TfrmCustomers, frmCustomers);
  Application.CreateForm(TfrmAbout, frmAbout);
  Application.Run;
```

The main form starts the server in `OnShow` instead of `OnCreate`, and stops it in `OnClose`. The comments explain why:

```
procedure TfrmMain.FormShow(Sender: TObject);
begin
  { Start listening. This is done in OnShow, not in OnCreate, on purpose.
    Pages.dpr creates this form first and then frmCustomers and frmAbout.
    Each of them plugs its DesignServer into HTTPServer while it is created.
    The main window is shown later, when Application.Run starts, so by then
    all three pages are plugged in and the first request finds them all. }
  HTTPServer.Active := True;
end;


procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  { Stop listening while the page forms still exist. They are freed after
    this window closes. }
  HTTPServer.Active := False;
end;


procedure TfrmMain.btnOpenClick(Sender: TObject);
begin
  { Opens the Home page in the default browser. The menu leads to the
    other two pages. }
  ShellExecute(Handle, 'open', 'http://127.0.0.1:8086/', nil, nil,
    SW_SHOWNORMAL);
end;
```

The two page units have no code. They only add `uMain` to their `implementation` section, so the IDE can show the link to `frmMain.HTTPServer` in the designer:

```
implementation

uses
  { The IDE needs it to show the link to frmMain.HTTPServer in the designer. }
  uMain;

{$R *.dfm}
```

### How it works

1. `Pages.dpr` creates `frmMain` first, because it owns the web server, then `frmCustomers` and `frmAbout`. The two page forms stay hidden: they only exist to be served.
2. While each form is created, its DesignServer reads the sgcHTML components on that form, the ones inside the frame included, and plugs itself into `HTTPServer`.
3. When the main window is shown, `FormShow` sets `HTTPServer.Active := True`. By then all three DesignServers are plugged in, so the very first request finds every page.
4. The browser asks for `/customers`. Every DesignServer sees the request, and only the one whose `PagePath` is `/customers` answers it with the Customers page. The Bootstrap files are served the same way as before.
5. You click a menu entry. It is an ordinary link, so the browser loads the next page, and that entry is highlighted because its form set `Active` on it.
6. An address that no DesignServer owns, such as `/nothing`, gets 404 Not Found.
7. Closing the main window sets `HTTPServer.Active := False` in `FormClose`, while the page forms still exist.

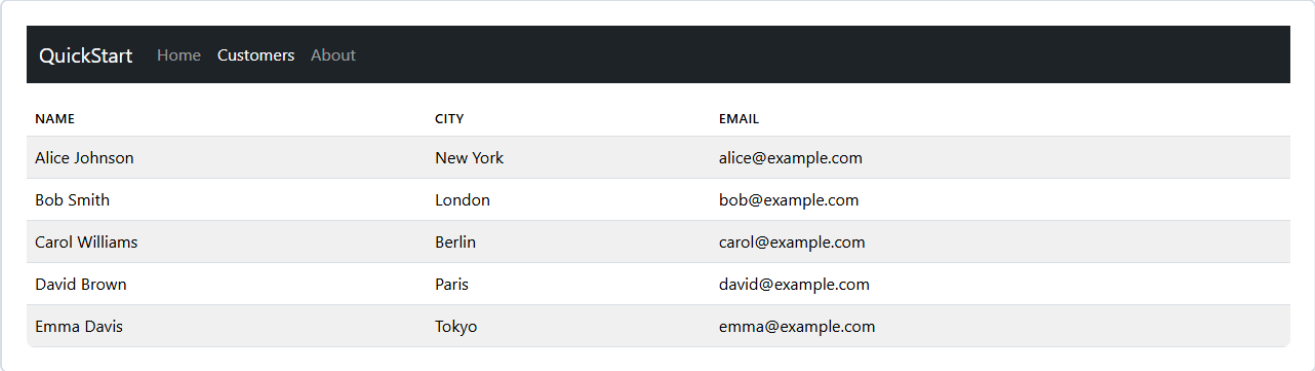



Figure 4.16. The Customers page at /customers.

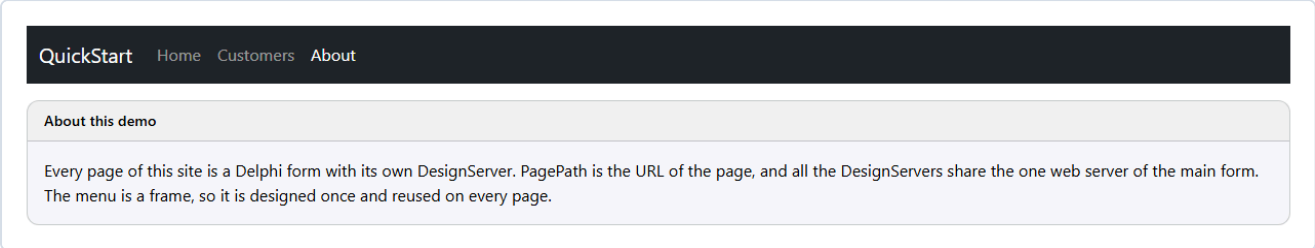



Figure 4.17. The About page at /about.

> **Watch out.** To link a page's DesignServer to `frmMain.HTTPServer`, open `uMain` in the IDE first. The Server list of the Object Inspector only offers components of forms that are open. The same goes for the menu: each page form stores its own copy of the `Items` collection, because that is where its `Active` entry is set. After you add or remove an entry in `uNavFrame`, make the same change on every page form.

## Try this

- Change `Brand` of the NavBar in `uNavFrame` to the name of your site. All three pages follow, because no page overrides `Brand`.
- Add a sixth customer to `Grid.Rows` on the Customers page, with a Tab between the name, the city and the email.
- Move the About page to `/info`: change `PagePath` of its DesignServer, then change the `Href` of the About entry in the menu copy of every page form.
- Add a fourth page. Add a new form to the project, drop a `TsgcHTMLDesignServer` (`Server = frmMain.HTTPServer`, a new `PagePath`), a `TsgcHTMLDesignSurface`, the `TfraNav` frame and a panel, then add an entry for it to the menu of every page. Check in `Pages.dpr` that the new form is created after `frmMain`.

# 5. Main components

The SGC HTML palette page holds 128 classes. You do not need most of them to start. This chapter introduces the sixteen that cover a typical first application, grouped by what they do. The Design Surface can lay out 111 classes, in seven categories: Layout (24), Input (24), Charts (22), Data (15), Media (10), Messaging (9) and Security (7). Once you know the components below, browsing those categories is the quickest way to find the rest.

Every class has its own help topic, named after the class: `TsgcHTMLComponent_Panel.htm`, `TsgcHTMLComponent_Grid.htm` and so on. Each topic lists every property with its type and default value.

## Reference table

| Group | Class | Purpose | Key properties | LoadFromDataSet |
|---|---|---|---|---|
| Layout | `TsgcHTMLComponent_Panel` | Card with title, body and footer | `Title`, `Body`, `Footer`, `Color`, `Collapsible` | no |
| Layout | `TsgcHTMLComponent_Tabs` | Tab strip | `Items` (`Title`, `Content`), `Style`, `Vertical`, `Fill` | no |
| Data | `TsgcHTMLComponent_Grid` | Data grid with sort, filter, paging and export | `Columns`, `Striped`, `Hover`, `ShowSort`, `ShowFilter`, `PageSize`, `DataSource` | yes |
| Data | `TsgcHTMLComponent_DataTable` | Table with search, row count, page size and export toolbar | `Title`, `ShowSearch`, `ShowPageSize`, `PageSizes`, `ShowExport` | yes |
| Data | `TsgcHTMLComponent_StatCard` | KPI tile | `Title`, `Value`, `Icon`, `Trend`, `TrendValue` | no |
| Data | `TsgcHTMLComponent_Chart` | Chart.js chart | `ChartType`, `Labels`, `Datasets`, `Title`, `ShowLegend` | yes |
| Feedback | `TsgcHTMLComponent_ProgressBar` | Progress bar | `Value`, `Max`, `Color` or `ColorStyle`, `Striped`, `ShowLabel` | no |
| Feedback | `TsgcHTMLComponent_Modal` | Bootstrap dialog | `Title`, `Body`, `Footer`, `Size`, `Centered` | no |
| Input | `TsgcHTMLComponent_Form` | Form built from a field list | `Fields` (`Name`, `FieldType`, `Label_`, `Required`, `Placeholder`, `FieldID`), `Method`, `Layout`, `SubmitText` | yes |
| Input | `TsgcHTMLComponent_Edit` | Labelled text input | `ElementName`, `Label_`, `Value`, `Placeholder`, `InputType` | no |
| Input | `TsgcHTMLComponent_Memo` | Multi-line text | `ElementName`, `Label_`, `Value`, `Rows` | no |

| Group | Class | Purpose | Key properties | LoadFromDataSet |
|---|---|---|---|---|
| Input | `TsgcHTMLComponent_Select` | Drop-down list | `Options`, `ElementName`, `Label_`, `Placeholder`, `Multiple` | yes |
| Input | `TsgcHTMLComponent_ButtonGroup` | Buttons; one item is a plain button | `Items` (`Text`, `ButtonStyle`, `Href`), `Size`, `Vertical` | no |
| Security | `TsgcHTMLComponent_Login` | Sign-in form or page | `FormAction`, `Title`, `ButtonText`, `ShowRememberMe`, `Auth` | no |
| Navigation | `TsgcHTMLComponent_NavBar` | Top navigation bar | `Brand`, `BrandHref`, `Items` (`Text`, `Href`, `Active`), `Theme` | no |
| Navigation | `TsgcHTMLComponent_Sidebar` | Side menu | `Items`, `Brand`, `Dark`, `CSSWidth` | no |

> **Note.** Several properties end in an underscore, such as `Label_`. `Label` is a reserved word in Pascal, so it cannot be used as a property name.

## Layout

### TsgcHTMLComponent_Panel

A Bootstrap card, and the component you will use most. `Title` is the header, `Body` the content and `Footer` an optional bottom line. `Body` accepts HTML, so `<p>` paragraphs and `<strong>` work there. `Color` tints the card, and `Collapsible` lets the user fold it away. You can drop other sgcHTML components inside a panel, as Demo 05 does with its forms, and they are nested inside the card on the page. Help: `TsgcHTMLComponent_Panel.htm`.

### TsgcHTMLComponent_Tabs

The web counterpart of a `TPageControl`. Each entry in `Items` is one tab, with a `Title`, a `Content` in HTML and an `Active` flag for the tab shown first. `Style` switches between classic tabs (`tsTab`), pills (`tsPill`) and an underline (`tsUnderline`). `Vertical` puts the tabs on the side, and `Fill` stretches them across the full width. Help: `TsgcHTMLComponent_Tabs.htm`.

## Data

### TsgcHTMLComponent_Grid

The workhorse for data. Bind it with `DataSource` and `DataAutoRefresh`, as chapter 3 explains, or fill it once with `LoadFromDataSet`. `Columns` chooses which fields appear and how: each column has a `Name` (the field), a `Title` (the header), an `Align` and flags such as `Sortable` and `Filterable`. `Striped` and `Hover` are on by default. `ShowSort`, `ShowFilter` and `ShowExport` add sorting, filtering and an export button, and a `PageSize` above zero splits the rows into pages. The grid has many more options, from inline editing to grouping and tree mode, which you can explore later. Help: `TsgcHTMLComponent_Grid.htm`.

### TsgcHTMLComponent_DataTable

A grid with a ready-made toolbar. `ShowSearch` adds a search box, `ShowPageSize` a page size selector whose choices come from `PageSizes`, and `ShowExport` an export button. `Title` is shown above the table. It contains its own `Grid` and `Pagination` objects, whose properties you can set too. Use it when users need to search and page through a list; use the plain Grid when you only need to show rows. Help: `TsgcHTMLComponent_DataTable.htm`.

### TsgcHTMLComponent_StatCard

A KPI tile: one number with a caption. `Title` is the caption and `Value` the number, as a string so you can format it yourself. `Icon` takes an HTML entity or a short piece of markup, `Trend` draws an arrow (`stUp`, `stDown`, `stNeutral`) next to `TrendValue`, and `FooterText` explains the trend. `Color` picks a Bootstrap color such as `scSuccess` or `scWarning`. A stat card is a good action target: change `Value` in `OnExecute` and only the card is redrawn. Help: `TsgcHTMLComponent_StatCard.htm`.

### TsgcHTMLComponent_Chart

A chart drawn by Chart.js in the browser, with the script served from your executable. `ChartType` picks the kind, from `ctLine`, `ctBar` and `ctPie` to more specialized types such as `ctWaterfall` and `ctFunnel`. `Labels` holds the categories, and each item in `Datasets` is one series with its `Label_`, its comma-separated `Data` and its `BackgroundColor`. Each label must be written in double quotes, as in `'"Mon","Tue"'`, which chapter 6 explains. `LoadFromDataSet` fills the chart from a dataset, given the name of the field that holds the labels. Help: `TsgcHTMLComponent_Chart.htm`.

## Feedback

### TsgcHTMLComponent_ProgressBar

A Bootstrap progress bar. `Value` out of `Max` sets how full it is. Pick its color with `ColorStyle`, a Bootstrap color such as `hcSuccess`, or with `Color`, a CSS color string. `Striped` and `Animated` add moving stripes, and `ShowLabel` prints a text inside the bar, built from `LabelFormat` (for example `'Monthly goal: %d%%'`). `CSSHeight` sets the thickness. Help: `TsgcHTMLComponent_ProgressBar.htm`.

### TsgcHTMLComponent_Modal

A Bootstrap dialog with `Title`, `Body` and `Footer`. `Size` chooses the width (`msSmall` to `msFullscreen`), and `Centered`, `Scrollable` and `StaticBackdrop` change how it behaves. The dialog stays hidden until a button or link that targets its `ModalID` opens it. For a simple yes or no question before an action runs, the action's `Confirm` property is easier. Help: `TsgcHTMLComponent_Modal.htm`.

## Input

### TsgcHTMLComponent_Form

A complete form, built from a list of fields. Each item in `Fields` has a `Name` (the key you read in `aAction.Params`), a `FieldType` (`ftText`, `ftEmail`, `ftNumber`, `ftTextArea`, `ftSelect`, `ftCheckbox`, `ftDate` and more), a `Label_`, a `Placeholder`, a `Required` flag and an optional `FieldID`. `SubmitText` is the caption of

the submit button. `Method` is `fmPost` by default, and `Layout` arranges the fields vertically, horizontally, inline or as a search bar. Pair it with an action whose `Source` is the form and whose `Trigger` is `dtSubmit`, as in Demo 04 and Demo 05. Help: `TsgcHTMLComponent_Form.htm`.

### TsgcHTMLComponent_Edit and TsgcHTMLComponent_Memo

Single inputs with their own label. `ElementName` is the name attribute of the input, the key its value is sent under when the browser submits it, `Label_` the caption above it and `Value` the initial text. The Edit adds `Placeholder`, `HelpText` and `InputType` (`itText`, `itEmail`, `itPassword`, `itNumber`, `itDate` and others), and the Memo adds `Rows` for its height. Both live in the unit `sgcHTML_Component_Edit`. Help: `TsgcHTMLComponent_Edit.htm` and `TsgcHTMLComponent_Memo.htm`.

### TsgcHTMLComponent_Select

A drop-down list, the counterpart of a `TComboBox`. Each item in `Options` has a `Value` (what is sent), a `Text` (what is shown) and a `Selected` flag. `ElementName`, `Label_` and `Placeholder` work as in the Edit, and `Multiple` turns it into a multi-select list. `LoadFromDataSet` fills the options from a dataset. Help: `TsgcHTMLComponent_Select.htm`.

### TsgcHTMLComponent_ButtonGroup

A row of Bootstrap buttons, and also the way to make a single plain button: give it one item. Each item has a `Text`, a `ButtonStyle` (`bsPrimary`, `bsSuccess`, `bsDanger` and the other Bootstrap styles) and an optional `Href` for a link. To run Delphi code on a click, make the ButtonGroup the `Source` of an action, as Demo 03 does. Set `AriaLabel` so screen readers can name the group. Help: `TsgcHTMLComponent_ButtonGroup.htm`.

## Security

### TsgcHTMLComponent_Login

A sign-in form with a user name field, a password field, an optional *Remember me* check box (`ShowRememberMe`) and a submit button (`ButtonText`). `LoginStyle` shows it as a card, a centered card or a full page. The form posts to `FormAction`, and the field names are fixed (`username`, `password` and `remember`) so the server always knows what to read. Assign `Auth` to a `TsgcHTMLAuth` to use the built-in sign-in, sessions and CSRF protection of the code-first path, described in `HTML_Sessions_Auth.htm`. Help: `TsgcHTMLComponent_Login.htm`.

## Navigation

### TsgcHTMLComponent_NavBar

The menu bar at the top of the page. `Brand` is the title on the left and `BrandHref` where it links. Each entry in `Items` has a `Text`, an `Href` and an `Active` flag that highlights the current page. `Theme` is `ntDark` by default, with `ntLight` as the alternative. On a narrow screen the items fold into a menu button. Demo 06 puts one NavBar on a frame and reuses it on every page. Help: `TsgcHTMLComponent_NavBar.htm`.

### TsgcHTMLComponent_Sidebar

A vertical menu for the side of the page, the usual layout of admin tools. `Items` are the entries, each with a `Text`, an `Href`, an optional `Icon` and an `Active` flag. Items can also be section headers (`Header`) or separators (`Divider`). `Brand` is shown at the top, `Dark` switches to a dark background and `CSSWidth` sets the width. Help: `TsgcHTMLComponent_Sidebar.htm`.

# 6. Common problems

These are the problems that came up again and again while the QuickStart demos were being built and tested. Each entry says what you see, why it happens and how to fix it. When something does not work, open the browser's developer tools (F12) and look at the Network tab first: a failed request usually points straight at one of the causes below.

## Setup and serving

### The page is unstyled

**What you see.** The page shows plain black text on white, with no cards, no colors and no columns.

**Why.** The page links `/bootstrap.min.css` and the browser could not load it. The DesignServer serves that file, and the scripts, only when it has a `PagePath`. They are also not served when you produce the HTML yourself, for example in your own `OnCommandGet` handler, or when that handler answers every request, including the ones for the style and script files.

**Fix.** Serve the page through a `TsgcHTMLDesignServer` with `PagePath` set, and let your own `OnCommandGet` handler leave unknown paths unanswered so the DesignServer can fill them. If you serve pages by other means, serve `/bootstrap.min.css`, `/bootstrap.bundle.min.js` and `/htmx.min.js` yourself.

### "Port already in use"

**What you see.** Setting `HTTPServer.Active := True` raises an exception saying that the socket could not be bound or that the address is already in use.

**Why.** Another program is already listening on that port. Most often it is a second copy of the same demo, still running in the background, or another web server such as IIS or a development server.

**Fix.** Close the other program, or change `Port` on the `TsgcHTTPServer`. To see which process holds the port, run `netstat -ano | findstr :8081` in a command prompt and look up the process id in the Task Manager. The QuickStart demos use the ports 8081 to 8086, one each.

### A page answers 404 right after the program starts

**What you see.** In a project with several page forms, the main page works but another page answers `404 Not Found`, at least for the first requests.

**Why.** Each DesignServer plugs itself into the web server while its form is created. A request that arrives before that form exists finds nobody serving its path. This happens when you start the server in the main form's `OnCreate`, because the other forms are created after it.

**Fix.** Start the server once all page forms exist. Demo 06 does it in the main form's `OnShow`:

```
procedure TfrmMain.FormShow(Sender: TObject);
begin
  { Start listening. This is done in OnShow, not in OnCreate, on purpose.
    Pages.dpr creates this form first and then frmCustomers and frmAbout.
    Each of them plugs its DesignServer into HTTPServer while it is created.
    The main window is shown later, when Application.Run starts, so by then
    all three pages are plugged in and the first request finds them all. }
  HTTPServer.Active := True;
end;
```

Also check that each `PagePath` is exactly the path you type: `'/customers'` is not `'/customers/'`.

### The project does not compile because of its name

**What you see.** A project called `Forms.dpr` fails to compile with an error about the name.

**Why.** The program name is an identifier too. A project named like a VCL unit it uses, such as `Forms`, collides with that unit.

**Fix.** Rename the project. The QuickStart forms demo is called `FormsDemo.dpr` for this reason.

### FireDAC complains about a missing object factory

**What you see.** Opening a FireDAC connection raises an error saying that the object factory for `TFDGUIxWaitCursor` is missing, or fails because it cannot load `sqlite3.dll`.

**Why.** A VCL FireDAC application needs its wait cursor implementation linked in, and the SQLite driver needs the SQLite engine, either as a DLL or linked statically.

**Fix.** Add the units that Demo 05 uses:

```
uses
  Windows, SysUtils, Classes, Controls, Forms, StdCtrls, ShellAPI, DB,
  { FireDAC: the components on the form and the SQLite driver, with the
    SQLite engine linked into the exe so no sqlite3.dll is needed. A VCL
    application also needs FireDAC.Comp.UI and FireDAC.VCLUI.Wait, or
    FireDAC raises "Object factory ... TFDGUIxWaitCursor is missing". }
  FireDAC.Comp.Client, FireDAC.Stan.Def, FireDAC.Stan.Async, FireDAC.DApt,
  FireDAC.Phys.SQLite, FireDAC.Phys.SQLiteWrapper.Stat, FireDAC.Comp.UI,
  FireDAC.VCLUI.Wait,
  // sgc
  sgcHTTP_Server, sgcHTMLDesign_Server_Indy, sgcHTMLDesign_Surface,
  sgcHTMLDesign_Actions, sgcHTML_Component_Grid, sgcHTML_Component_Form,
  sgcHTML_Component_Panel;
```

`FireDAC.Phys.SQLiteWrapper.Stat` links SQLite into the executable. The demo also sets `ResourceOptions.SilentMode` to `True` on the connection, so FireDAC never tries to show an hourglass cursor from a server thread.

## Events and threads

### Nothing happens when I click

**What you see.** You click a button or submit a form in the browser and nothing changes. Your breakpoint in `OnExecute` is never hit.

**Why.** The click never reaches an action. There are three usual causes: the action's `Source` is not the component you click, the `Trigger` does not match (a form needs `dtSubmit`, a button `dtClick`), or the browser could not load `/htmx.min.js`, so nothing sends the request.

**Fix.** Open `Surface.Actions` and check each item: `Source`, `Trigger`, `Enabled` and an assigned `OnExecute`. Then check the Network tab in the browser: `/htmx.min.js` must load with status 200. If the handler runs but the page does not change, check the `Target`: it must be the component whose properties the handler changed.

### Changing a VCL control in OnExecute crashes or freezes the program

**What you see.** The program freezes, raises odd access violations or paints garbage after a click in the browser, typically when the handler updates a `TLabel`, a `TMemo` log or shows a message box.

**Why.** `OnExecute` runs on a server thread. The VCL is not thread safe and must only be used from the main thread. The DesignServer lock protects the sgcHTML components, not the desktop window.

**Fix.** In `OnExecute`, change only sgcHTML components and your own data. Never call `ShowMessage` there: it would run on a server thread, and the person in the browser would never see it anyway. If the desktop window really must reflect something, hand the update to the main thread without waiting, with `TThread.Queue`.

### One user's changes appear for everyone

**What you see.** A value one visitor produced, such as a result panel or a counter, shows up in another visitor's browser after a reload.

**Why.** One designed form is one page, shared by every browser. This is how the zero-code path works, not a bug. See chapter 3.

**Fix.** Keep shared pages for shared information: dashboards, status boards, kiosks and single-user tools. For per-user data, use sessions and sign-in, described in the help topic `HTML_Sessions_Auth.htm`.

### Text with < or & breaks the page

**What you see.** A name typed by a user shows up in bold, disappears, or a pasted `<script>` tag runs in the browser.

**Why.** `Body` and many other text properties are raw HTML. The browser reads whatever you assign as markup.

**Fix.** Pass every value that came from the browser through `sgcHTMLEncode` (unit `sgcHTMX_Helpers`) before you put it in HTML, as Demo 04 does. For SQL, always use parameters.

## Forms

### The form keeps the typed values after a submit

**What you see.** After a successful submit, the fields still show what the user typed.

**Why.** Only the action's `Target` is swapped. The form is not the target, so the browser keeps it as it is. This is on purpose: the user can fix a typo and submit again.

**Fix.** If users need a quick way to start over, set the form's `ShowReset` to `True`. It adds a reset button, with `ResetText` as its caption.

### Two forms interfere with each other

**What you see.** On a page with two forms that share a field name, clicking a label focuses the field in the other form, or the browser reports duplicate ids.

**Why.** Without a `FieldID`, the HTML id of a field is `field_` followed by its `Name`. Two forms with a field called `name` both produce `id="field_name"`, and the label of the second one points at the first.

**Fix.** Set `FieldID` on the fields to a unique value per form. Demo 05 uses `add-name`, `edit-name` and so on, while the `Name` stays `name` so the handlers read the same key from `aAction.Params`.

### ftDate is the wrong type in code

**What you see.** Assigning `ftDate` or `ftTime` to a form field's `FieldType` in code fails with an incompatible types error.

**Why.** The unit `DB` (`Data.DB`) declares `ftDate` and `ftTime` too, for `TFieldType`. When both units are in the `uses` clause, the one listed last wins.

**Fix.** Qualify the value with its unit: `sgcHTML_Component_Form.ftDate`. Values set in the Object Inspector are not affected.

## Layout and appearance

### Chart labels are missing or the chart is empty

**What you see.** The chart has no categories on its axis, or it does not draw at all.

**Why.** `Labels` is written straight into a JavaScript array. Without quotes, `Mon,Tue` is not valid text for the browser.

**Fix.** Write every label in double quotes, separated by commas, as in Demo 02:

```
ChartType = ctBar
Labels = '"Mon","Tue","Wed","Thu","Fri","Sat","Sun"'
```

The values in a dataset's `Data` are numbers, so they need no quotes.

### The chart is too big or too small

**What you see.** A chart in a wide column is very tall, or a chart refuses to fill the height you gave it on the form.

**Why.** A chart keeps its shape as it grows and shrinks. The bar chart of Demo 02 is twice as wide as it is high, and its width follows the Bootstrap column. The height of the component on the form is not used.

**Fix.** Make the column narrower, or set `CSSHeight`. Because the shape is kept, a smaller `CSSHeight` also limits how wide the chart gets.

### CSSClass or Style has no effect

**What you see.** You set `CSSClass` or `Style` on a ProgressBar, an Edit or a Select and nothing changes on the page.

**Why.** Every component inherits these two properties, but these three do not use them when they render.

**Fix.** Use the component's own properties instead, such as `CSSHeight` and `ColorStyle` on the ProgressBar. For spacing between rows, set `Surface.RowCSSClass`, which applies classes to every generated row.

### Extra space around a full-width component

**What you see.** A navigation bar or progress bar that spans the whole width has a gap or margin you did not ask for.

**Why.** Each row is wrapped in a Bootstrap row and column, which adds spacing, even when the row holds a single full-width component.

**Fix.** Set `Surface.CompactRows` to `True`. It drops that wrapper for rows holding one full-width component.

# 7. Where next

Once the six QuickStart demos feel familiar, you know the core of the zero-code path. This chapter points you to the larger demos, to the code-first path for applications that need routing, sessions or live updates, and to the documentation and support.

## The design-time demos

`Demos\60.HTML\02.DesignTime` holds 22 complete applications whose pages are designed on forms with a `TsgcHTMLDesignSurface`. They go further than the QuickStart: several of them serve their pages with their own code, using routes, the HTMX engine or WebSocket push. Read one next to its `readme.html`, which every demo folder includes. Good places to start:

| Demo | What it shows |
|---|---|
| `01.ERP` | An ERP designed on forms: several business pages behind one menu. |
| `05.HTMX` | A showcase of the htmx features, designed on forms. |
| `06.Grid` | The features of `TsgcHTMLComponent_Grid`: sorting, filtering, paging, export and more. |
| `08.Components` | A showcase of the sgcHTML components, a good catalog to browse. |
| `14.POS` | A retail point of sale, with its pages designed on forms. |

The others cover an admin CRUD (`02.AdminCRUD`), a live monitor (`03.LiveMonitor`), a customer portal (`04.Portal`), site layouts (`07.Site`), a helpdesk (`09.Helpdesk`), a shop assistant (`10.ShopAssistant`), two WebBroker mini CRMs (`11.WebBroker`, `12.WebBrokerHTML`), warehouse management (`13.Warehouse`), a BI reports portal (`15.Reports`), a multi-tenant SaaS (`16.SaaS`), field service (`17.FieldService`), the AI data pack (`18.AIData`), MCP apps (`19.MCPApp`), industrial instruments (`20.Instruments`), SCADA with MQTT (`21.SCADA_IoT`) and right-to-left pages (`22.RTL`).

## The run-time demos

`Demos\60.HTML\01.RunTime` has 22 demos with the same names, where the pages are built in code instead of on forms. Most of them are console servers that print their own URL when they start. Compare a run-time demo with its design-time twin to see how a component configured in the Object Inspector looks when you write it in Pascal.

## The code-first path

Everything in this guide used the zero-code path. The older code paths remain fully supported, and you will meet them in the larger demos:

- **`TsgcHTMLPageBuilder`** and the components created in code. You set properties and read the `HTML` property to render the page, then send it from your own request handler.

- **`TsgcHTMX_Engine_Server`**, the HTMX engine. It adds WebSocket push, for example `BroadcastFragment` to update a part of the page in every connected browser, and it applies sessions and sign-in through its `Auth` property.
- **`TsgcHTMX_Router`**, for routes with path parameters, methods and access rules.
- **Templates** such as `TsgcHTMLTemplate_Bootstrap` and `TsgcHTMLTemplate_HTMX`, for a page shell shared by many pages.

Both paths use the same components and produce the same markup, so what you learned here carries over.

## Help topics

The sgcWebSockets help has a section for sgcHTML. These topics are the natural next reads:

| Topic | What it covers |
|---|---|
| `HTML.htm` | The overview: how sgcHTML works and the component families. |
| `HTML_RuntimeVsDesignTime.htm` | Building pages in code or on a Design Surface, and when to use each. |
| `HTML_DesignInBrowser.htm` | The Web Design tab, Preview in Browser and Design in Browser. |
| `HTML_Routing.htm` | Routes, path parameters, methods, precedence and access rules. |
| `HTML_Sessions_Auth.htm` | Sessions, sign-in, CSRF protection and remember me. |
| `HTML_CRUD.htm` | CRUD pages, validation and the write path to your data. |
| `HTML_Transport.htm` | WebSocket or server-sent events for pushes and broadcasts. |
| `HTML_Navigation.htm` | Navigation between pages and live updates. |
| `TsgcHTMLComponent_<Name>.htm` | One topic per component, with every property, its type and its default. |

## On the website

- https://www.esegece.com/products/sgchtml/: the product page, with features, components and the gallery.
- https://www.esegece.com/products/sgchtml/quickstart/: the online quick start, including the code-first path.
- https://www.esegece.com/products/sgchtml/migrate/unigui/: a guide for moving from uniGUI, with the differences in more depth.

## Getting help

If you are stuck, open a ticket in the helpdesk of your account at https://www.esegece.com/. It helps to include your Delphi version, your sgcWebSockets or sgcHTML version, the demo or a small project that shows the problem, and what the browser's developer tools show in the Console and Network tabs.

Enjoy building your first web application in Delphi.

© 2026 eSeGeCe. sgcWebSockets and sgcHTML are products of eSeGeCe. https://www.esegece.com