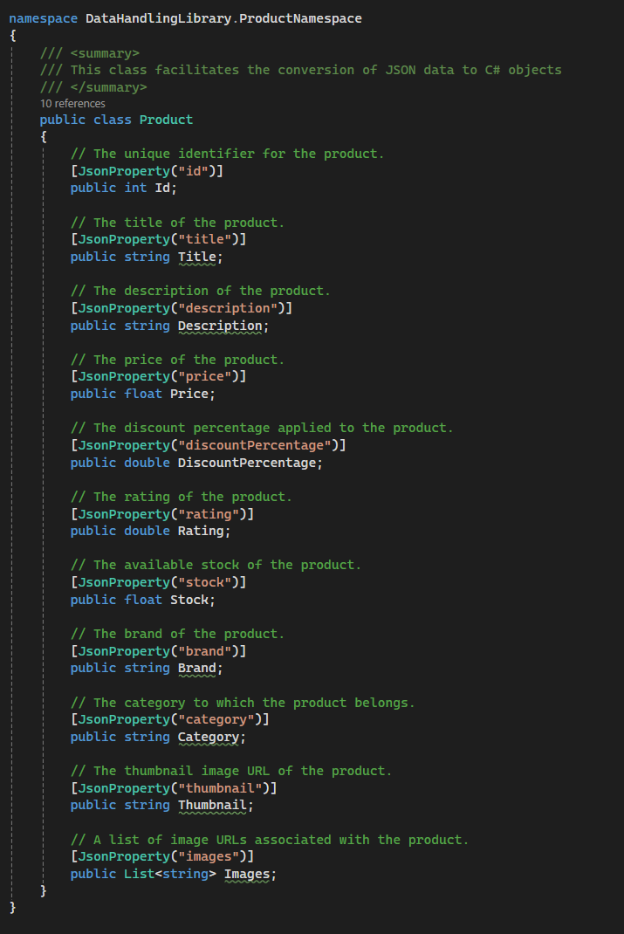


**CacheProvider (DataHandlingLibrary)**

| **Goal** | **How the code fulfils it** |
| --- | --- |
| **Stop hammering the API** – reuse data that rarely changes for a few minutes. | A private MemoryCache instance (from System.Runtime.Caching) is spun up with its own name, so every consumer gets an isolated cache store. |
| **Keep the library UI-agnostic** – cache logic lives in a class library, not in the WPF/Xamarin pages. | CacheProvider implements ICacheProvider, meaning the UI can talk to an *interface* and never worry about the implementation. If you later switch to Redis or LiteDB, you swap classes without touching the pages. |
| **Avoid stale objects** – every record should self-expire. | The constructor accepts dataLifeTimeMinut (default 10). AddDataToCache() stores the entry with an absolute expiration of DateTimeOffset.Now.AddMinutes(\_dataLifeTimeMinut). |
| **Defensive programming** | \* Negative lifetimes are clamped back to the default.  \* CheckStringValue() (omitted in the screenshot) throws if the cache name or key is null/empty, protecting against silent failures. |

**How it plays out at run-time**

1. A page (e.g. *ProductsReview*) asks Responses.GetProducts() for data.
2. Responses calls CacheProvider.AddDataToCache(); if it returns false, the item is already present and is read back with GetCachedData<T>() (next method in the file).
3. Only when the key is missing or expired does the code call ResponseHandler.SendGetRequestAsync() to hit the DummyJSON endpoint.  
   This shrinks API chatter and makes scrolling through long lists buttery-smooth.

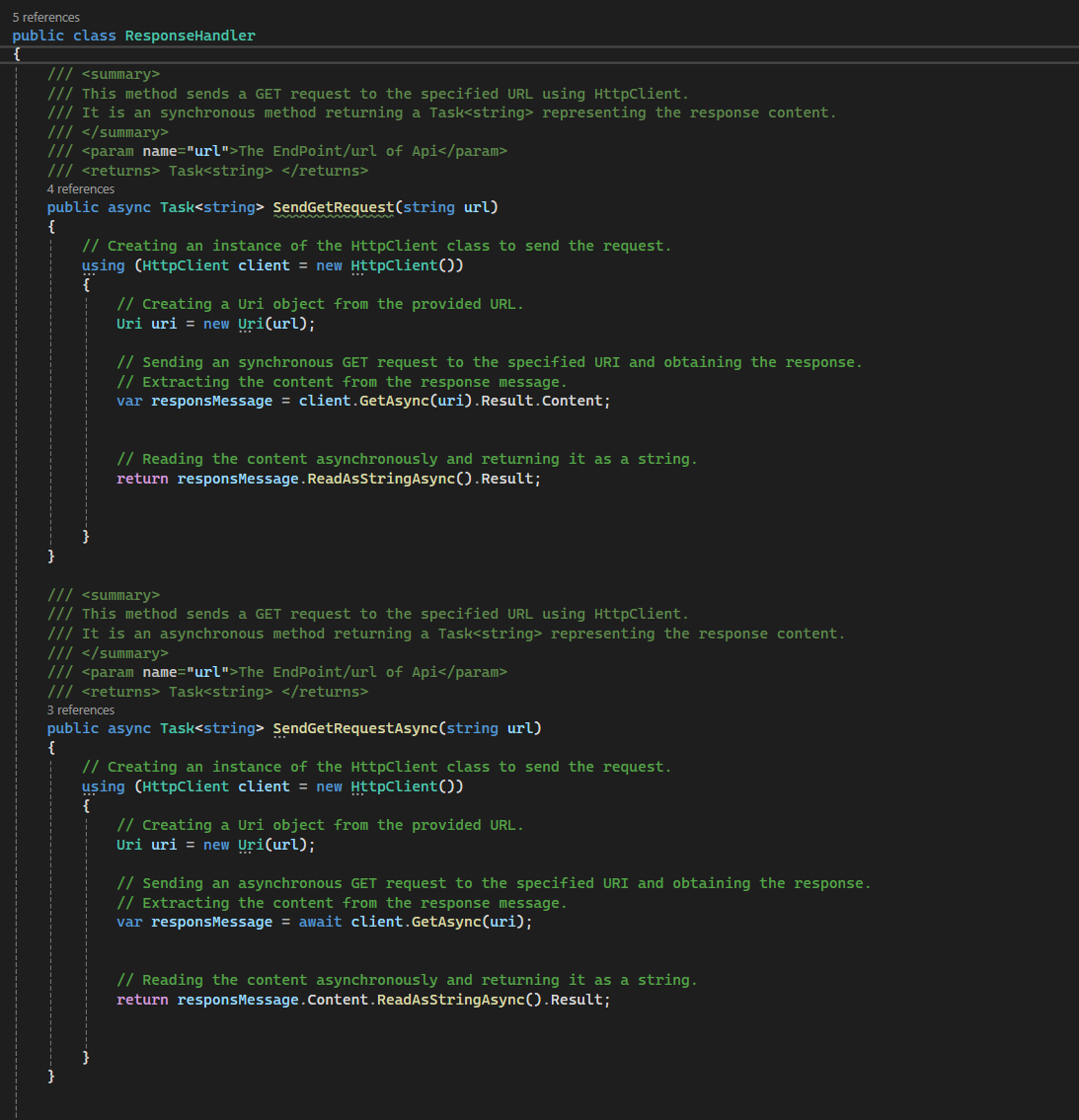


**Product DTO (DataHandlingLibrary.ProductNamespace)**

| **Design choice** | **Rationale** |
| --- | --- |
| **Flat POCO with public fields** (instead of properties + getters) | These objects are *pure data carriers*. By stripping away behaviour we minimise memory overhead and let Newtonsoft.Json populate them lightning fast via reflection. |
| **[JsonProperty("jsonKey")] attributes** | The DummyJSON API uses camel-case (discountPercentage); our C# naming can stay PascalCase while serialization still lines up 1-to-1. |
| **Strong typing for prices / ratings** | float/double let charts on *PriceMonitoringPage* plot numeric series without conversions; casting happens only once during deserialization. |
| **Kept in its own namespace** | Separating Products, Posts, Users, Comments into distinct namespaces keeps IntelliSense tight and avoids accidental cross-use. |

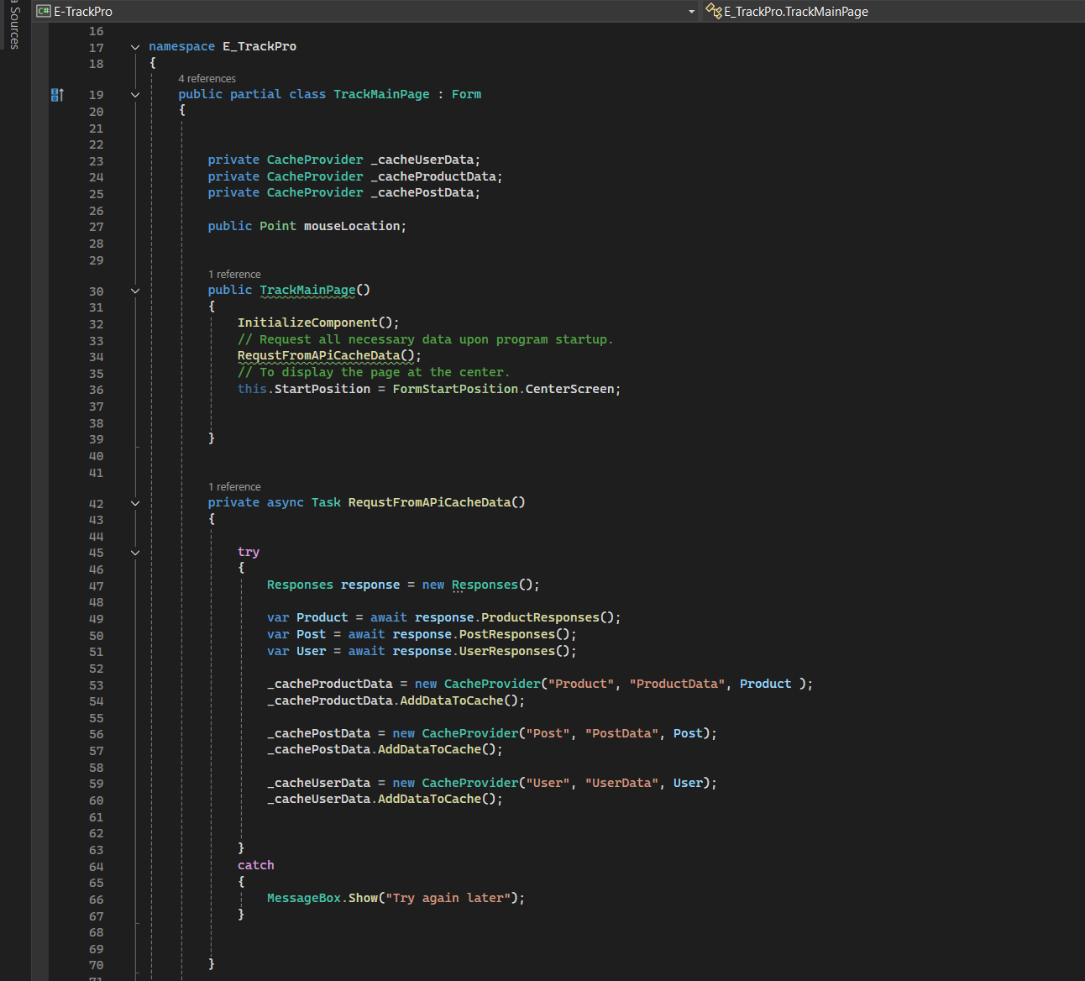
**Why this matters in practice**

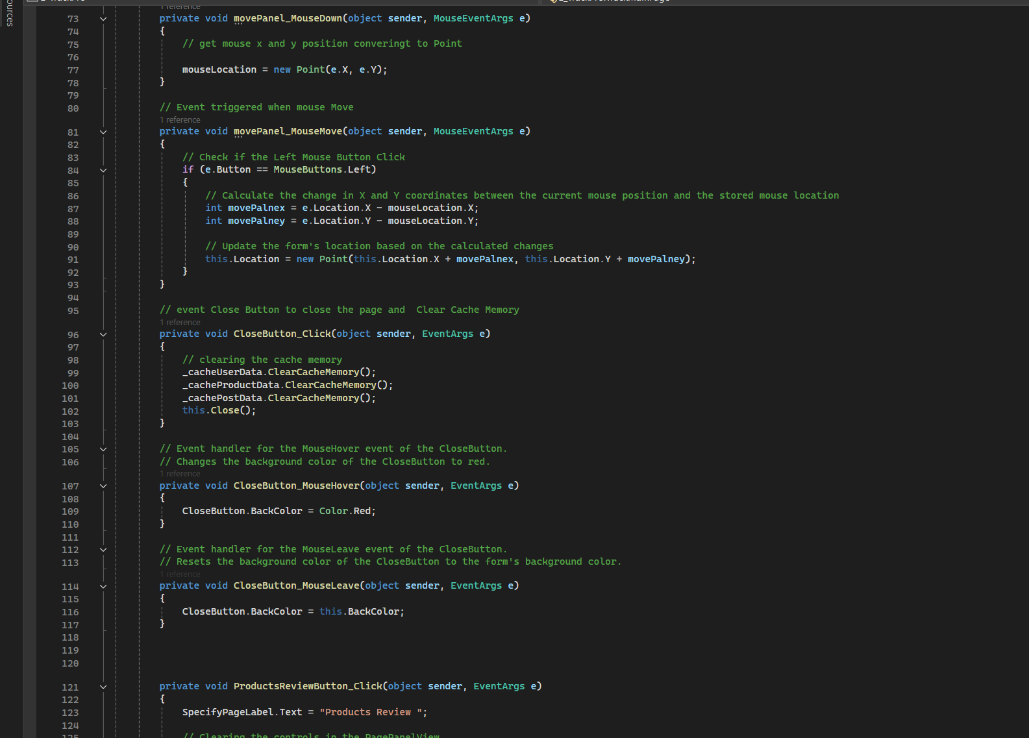
* The list cell templates (ListItemCell) receive a fully populated Product object and can bind straight to Title, Price, Rating, etc. – no extra mapping layer.
* When you PATCH-update a user, a *different* DTO (User) is used, but the pattern is identical, so new entities can be added in minutes.



**ResponseHandler – tiny HTTP wrapper**

| **Why wrap HttpClient?** | **What the code does** |
| --- | --- |
| **One exit point for every HTTP request** – easy to add logging, retry, auth headers later. | SendGetRequest (sync) & SendGetRequestAsync (async) each: 1) spin up a disposable HttpClient; 2) build a Uri; 3) fire GetAsync; 4) read the body text. |
| **Async all the way for UI threads** | Pages normally call the async version so the WPF dispatcher/Xamarin main thread never blocks. The sync flavour is there for unit tests or console tools that can afford blocking. |
| **Pure string IO** | The method returns raw JSON so DeserializeJsonData<T>() (next in the file) can funnel it into *any* DTO you pass – products, users, posts, comments. |





The “shell” of the whole app

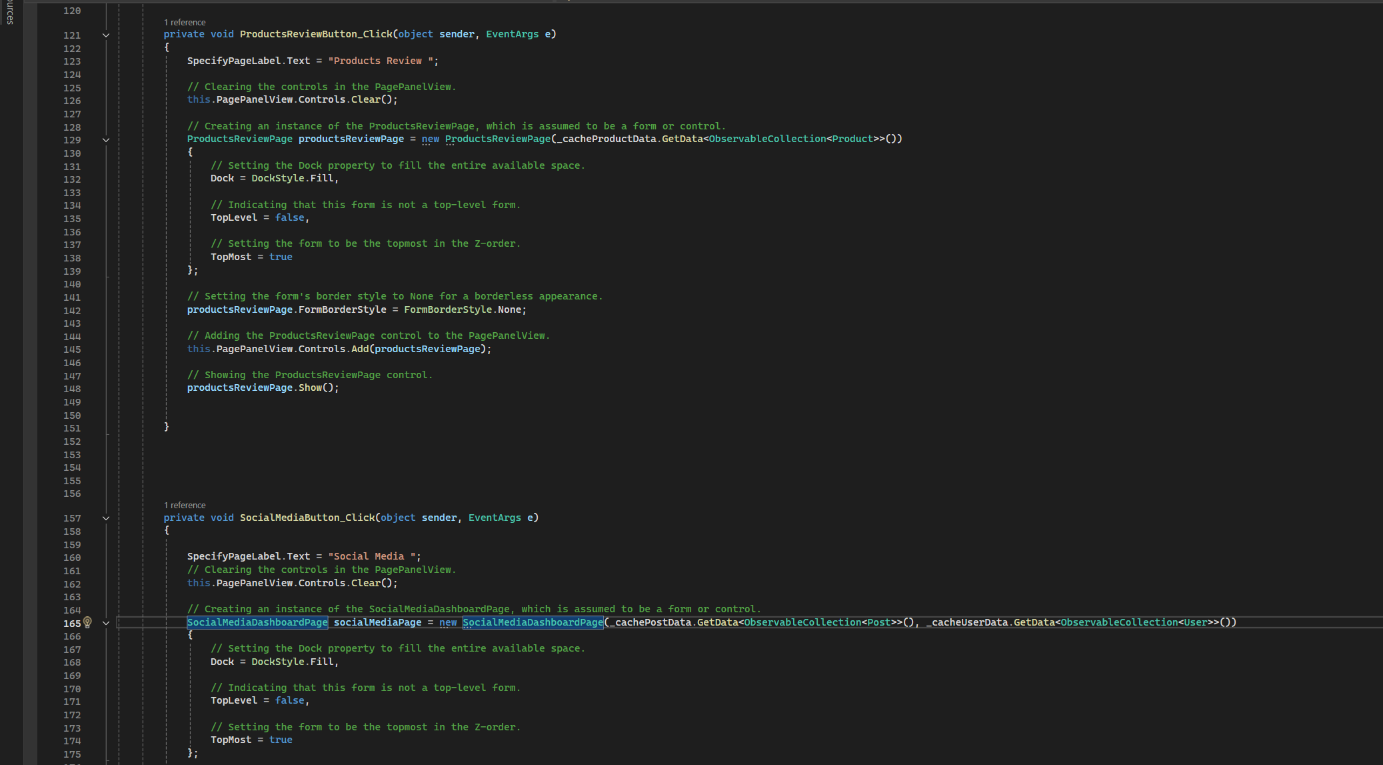
**WinForms as the host** – You chose Windows Forms because it spins up quickly, runs everywhere a .NET runtime exists, and its designer lets you lay out controls in minutes. This matters for a demo project whose goal is to *show API integration*, not UI-framework wizardry.

**partial class** – One half is the designer-generated code (TrackMainPage.Designer.cs), the other half (this file) is your **logic layer**. Keeping them split means the designer can regenerate layout code without stomping on your hand-written routines.

**Why load everything up-front?**  
Products, posts and users change rarely in DummyJSON. Grabbing them once keeps all subsequent page loads instant and offline-friendly.

**Why await three separate calls?**  
Parallel calls would save ~100 ms but add complexity. Serial awaits keep the sample clear while still being non-blocking (the UI thread stays responsive thanks to async/await).

**Why wrap raw objects in CacheProvider?**  
You already saw earlier how this class gives you TTL control and isolates cache logic from UI code



* **Why clear then add?**  
  PagePanelView is a placeholder panel. Clearing it first avoids multiple pages stacking up.
* **Pass cached data → constructor** – The page never calls the API; it simply binds to the already-deserialized ObservableCollection<T>.
* **Embed a *Form* inside a panel** – Setting TopLevel = false turns a Form into a control. This is a handy WinForms trick to get full designer support for each page without writing custom UserControls.
* **Dock = Fill** – The child page stretches to occupy the remaining area, leaving the left nav rail intact.

Identical logic repeats for the Social Media dashboard and the other tabs, each time passing the right slice of cached data.

* **Cache purge** – Clearing before Close() guarantees no lingering static MemoryCache items if you reopen the app in the same process (useful during debugging).
* **UI feedback** – Hover/leave events repaint the close icon red on hover, reinforcing that it’s clickable and dangerous.