Preserving your investment in SharePoint development

A lot of the appeal of SharePoint is that you can do so much without writing code. But no product can possibly meet everybody’s needs out of the box, and it’s good to know that if you hit a wall and simply can’t do something using out-of-the-box features and no-code development (like PowerApps and Flow), you can extend the product with code.

Microsoft has changed the rules a lot over the years, however, and this understandably gives pause to anyone thinking about investing in custom development on SharePoint.

|  |  |  |
| --- | --- | --- |
| Approach | Introduced in | Current status |
| Farm solutions | SharePoint 2007 | Works on premises  Not available online |
| Sandboxed solutions | SharePoint 2010 | Works on premises Not available online |
| JSLink, Display Templates | SharePoint 2013 | Works on classic sites only |
| SharePoint hosted add-ins | SharePoint 2013 | Still works everywhere but somewhat limited capability |
| Provider hosted add-ins | SharePoint 2013 | Still works everywhere but based on an outmoded authentication technology (ACS) |
| “Remote provisioning” (pushing JavaScript onto site pages) | n/a | Works on classic sites only |
| SharePoint Framework | SharePoint Online early 2016 | Works on premises in SP2016 FP2  Works online |

So how can you develop for SharePoint with some confidence that Microsoft won’t change the rules again? Here are my thoughts – speaking as an individual who’s been in this game for a long time, and not on behalf of Microsoft (even though I work for them, my job doesn’t give me visibility over that kind of thing anyway!) And, for what it’s worth, I’d apply these principles to any SaaS product that allows customization, because they’re all dealing with the same changing technology landscape.

## The web browser is a great common denominator

As much as SharePoint has changed over the years, one big constant has been that it runs in a web browser. Web browsers have changed too, but they still use HTML and JavaScript, and there’s always a lot of backward compatibility due to the number of web sites already out there.

For example, there were two ways to write a sandboxed solution: put the code in the server-side sandbox, or put the code on the web page and run it in the browser. When sandboxed solutions went away, those who put the code on the web page could reasonably port it to a SharePoint Hosted Add-in, the SharePoint Framework, or even a Content Editor Web Part. Those who depended on the sandbox inside of SharePoint were looking at a complete rewrite.

As an example, figure 1 shows a “future proof” site creation web part I demonstrated at SPTechCon 2010; it ran on the client side in a content editor web part. Figure 2 shows the same web part running in the SharePoint Framework at SPTechCon last year. While there was certainly some repackaging work to do, 100% of the code was preserved because it only relied on the web browser and the SharePoint API.

[figure 1 – Site creation web part in SharePoint 2010]

[figure 2 – Site creation web part in SharePoint Framework]

Another example is this [header and footer](https://blogs.msdn.microsoft.com/bobgerman/2018/03/11/building-headers-and-footers-that-work-on-classic-and-modern-sites/) which run in both classic and modern sites, by virtue of the fact that they only depend on a web browser and JavaScript in order to work.

Bottom line: it’s always safer to run code outside of SharePoint, on the browser or elsewhere, than inside of some product tooling like a sandbox or customizable web part.

## Depend on APIs, not internals

In fairness, my examples depended on more than just the web browser, I had to use SharePoint API’s to get at the content. Unlike the development models, SharePoint’s API’s haven’t changed much over the years.

|  |  |  |
| --- | --- | --- |
| API | Introduced in | Current status |
| SOAP web services | 2007 | Officially deprecated but still works online and on prem |
| CSOM/JSOM | 2010 | Fully supported online and on prem |
| REST via listdata.svc | 2010 | Fully supported online and on prem |
| REST via \_api | 2013 | Fully supported online and on prem |

Part of the problem with SharePoint development is that we’ve often taken dependencies on product internals. For example, back in SharePoint 2003 when we moved our code into the SharePoint installation directory, our work could easily be overwritten by a service pack. Or, even today, when code assumes certain HTML elements will always be part of a page or list form, the HTML element we want might just go away without notice. This was tolerable on premises, where we could test all our code with each new service pack and release before installing it in production, but it’s a big risk online where changes are continual.

If your code depends on something internal, Microsoft might break it without even knowing. On the other hand, Microsoft thinks long and hard before it changes a supported API. Stay with published API’s and your code is much less likely to become obsolete!

## Follow the Money

In SharePoint online, Microsoft can monitor feature usage in ways that were never possible on premises. This telemetry allows them to make business decisions on what features to keep and which it can shut off. It starts with usage but it boils down to money – how much customer investment, and how much of Microsoft’s own investment, is hanging on a particular feature.

In general, if you’re going to depend on a SharePoint feature, think about this:

* Features that are obscure and hardly anybody uses are at the greatest risk of finding their way to the chopping block. (The SharePoint Design Manager is a good example.)
* Features that are popular, so millions of users are counting on them, are at much less risk. Even if Microsoft wants to shut them off they will give years advance notice (InfoPath is a good example.)
* Features that Microsoft itself depends on, so other products and features would break without them, are at the least risk of all (The SharePoint Framework is a good example.)

## Where does this leave us?

All the current best practices are based on these principles. The PnP community has created a bunch of tools that run outside of SharePoint and depend only on supported API’s; anything from PnP is on the safe list. Also, the SharePoint Framework and Add-in model are well adopted and supported, and again, they only depend on API’s and the web browser, so even if they changed, you could port the code into some other model. Older, more arcane technologies like JSLink, display templates, XSLT web part customization etc. are all much riskier at this point, because they’re fairly obscure and tied to arcane bits of classic SharePoint.

If you want to learn more, please attend SPTechCon Boston 2018. One cool thing about SPTechCon is that it’s not only focused on futures, it provides sessions that address people where they are right now, which often means running on premises with a mix of older versions of SharePoint. While the sessions haven’t been finalized at the time of this writing, there’s a good chance Julie Turner and I will be presenting some “future-proof” themed topics that help you develop for the SharePoint you have today and preserve your investment when you move to modern sites in the future.