Shortening the List
How to Choose a Component Library

Peter Vogel

WHITEPAPER

About the Author

Peter Vogel

Peter Vogel is a system architect and principal in PH&V Information Services. PH&V provides full-stack consulting from UX design through object modeling to database design. Peter also writes courses and teaches for Learning Tree International.

Developers tend to look at applications, ask “How do I build that?” … and then go on to build the whole thing. However, developers are most productive when they’re working on delivering the functionality that’s unique to your application. Developers’ productivity is actually negative when they’re “re-developing” functionality that either already exists in the organization or can be acquired from some other source.

It gets worse: the more complex the functionality, the longer it takes to redevelop that functionality … which means, of course, that your developers’ productivity just keeps getting lower as your applications become more complex. And that’s a problem because modern business applications need complex functionality—everything from data grids that work with a wide variety of data sources, to UI components that provide rich interactions on all platforms, to sophisticated visualizations that enable users to make good business decisions quickly. In order to deliver applications efficiently, developers need a library of reusable components that handle those complex but common tasks so that developers can concentrate on creating the functionality that’s unique to your organization’s applications.

But, while reuse can avoid those productivity losses, as we all know, reuse isn’t free, even when the purchase price for a component is zero because not all components are created equal. If you pick the wrong component, developers lose time trying to get a recycled component to work as required (potentially even discovering the component “can’t do that”). Even if a developer can, in the end, make a poor component work, that unfortunate component is now embedded in the application, continuing to soak up developer time by driving up maintenance costs.

So, the costs associated with acquiring and using third-party components are real. Those costs need to be offset by saving time either in development or in making extensions/enhancements/bug fixes after deployment (ideally, in both places). Picking the right component can even pay for itself with improved quality (e.g., with fewer bugs or enabling a user experience that improves user performance).

More critically, acquiring a component is a long-term investment: You want to use any component for a long period of time because, over time, developers get better and better at using it, further increasing productivity. It's a decision worth spending time on.

Your first decision in this area is whether you want to acquire individual components on an "as needed" basis or get a component library. Acquiring single components reduces your cost for any purchase and, therefore, your risk (though you're really just spreading the risk out because, at some point, it would have been cheaper to buy a library).

Acquiring a component library has four key benefits, two of which are related to productivity:

- Encouraging reuse by giving developers access to a pool of ready-to-use components that developers can access easily
- Improving productivity through knowledge transfer (i.e., if a developer knows how to use one component in a library, they know a great deal about using any other component)
- Improving productivity through synergy because components in a library typically “play well together”
- Reducing risk by making one good decision rather than a series of decisions where each decision could be … unfortunate

But there’s a good news/bad news story here: The good news is that you have a lot of libraries to choose from … which is, of course, also the bad news because you only want to pick one. Furthermore, this is not a one-size-fits-all solution—you have to pick the right library for your organization.

You need some set of criteria that will reduce the risk of making the wrong decision by filtering the list of libraries to, ideally, a list of one. These are the criteria that you can select from to make the right decision for your organization.

First Choices

There are three choices that, while they might shorten the list, can lead you astray: cost, familiarity and popularity.

Cost: There are many libraries that are free to purchase and free does reduce part of your risk. But you need to consider whether the library has all the functionality you need, the level of knowledge transfer and synergy it provides and, since this is a long-term decision, the “total cost of ownership.”

Familiarity: It can be tempting to pick a library from a source that your organization has already got products from (perhaps even other components). Familiarity is tempting because it’s an excellent criteria—there are real benefits to developer productivity in getting components that “work the same” as components developers already know. However, you want to make sure that library’s components really do “work the same,” aren’t missing functionality you need and won’t incur avoidable costs. Sometimes you’ll need to start fresh.

Popularity: This is another great criteria—there are good reasons why a component library is popular. Unfortunately, those may not be the reasons that make sense for you.

So, what criteria should you prioritize?

Business-Specific Needs

Chief among the criteria that will reliably reduce the list for you are any specific needs that your organization’s applications must have and that only some component libraries deliver.

If, for example, your organization is working with geographic data, then you’ll need components that support geospatial data and rendering. If your organization is in a financial segment, you may need components that support in-depth financial analysis beyond what most libraries will support. If that doesn’t sound like you, you can skip the rest of this section.

If that does sound like you, then the reality is that sources that provide those specialized components probably don’t provide components that meet more common needs. That being the case, you’ll want to pick the right specialized component library first and, after that, pick another library with “every other component” that’s compatible with your specialized library.

But if all the “other libraries” are equally compatible, then you’ll need to look at additional criteria to shrink your list.

Supporting Multiple Platforms

Your next criteria should be the platforms your applications will run on. If you’re only building web applications or only targeting the Android platform, you should be looking only at libraries for that platform. Unfortunately, that’s less help than you might hope for: There are typically a lot of libraries on the list for any particular platform.

If you’re supporting multiple platforms and using a cross-platform development tool (for example, Microsoft’s .NET MAUI platform), you’ll only be looking at libraries that support your development tool. That could shorten the list of libraries that you’ll look at: Often, there’s only a small number of libraries that support any particular cross-platform tool.

However, the typical case is you’re supporting multiple platforms by using multiple platform-specific development toolsets so you’ll need a library for each platform. You should shorten your list by looking at vendors that provide support for all (or most) of the platforms you intend to support.

Cost matters here: Can you acquire components for just the platforms you support or do you have get “all the platforms”? After all, if it’s going to cost you more, there’s no point in getting components for a platform you *don’t* support. Coming at this from the other end of the problem: Is there a discount for “buying in bulk”? Can you get support for all the platforms you need for less than buying several platform-specific libraries?

Building an Inventory

Hopefully, these criteria have already shortened your list. But now you need to consider your organization’s specific needs.

It’s a good idea at this point to take a few minutes and make a list of your current applications—what you’ve built in the past is a pretty good indicator of what you’ll build in the future. With that high-level inventory in hand, take into account what you wish you *could have done* with those applications, and add in what you’d like to do in the future. You’ve now built an inventory of your required functionality, some of which can be met by a component library.

In that inventory you’ll probably find that, sometimes, you have to deliver an application that will be used infrequently, only by company staff and only to perform some simple task (updating a list of company departments, for example). What matters in that scenario is delivering the application quickly because, for example, almost any UI will be good enough (and “good enough” is good enough in this scenario).

On the other hand, sometimes you need to deliver an application that will be used frequently by people outside the company (customers, business partners, etc.) to do something that’s mission critical. In this scenario, building the right application is going to be essential—getting this application wrong can literally cost your company business. These are the “business critical applications.”

As you build your inventory, you’ll also need to consider the backend tools that your components will depend on. The obvious example here are the tools you’re using to store (and deliver) your organization’s data—any components you get will need to work with those tools.

Less obvious is reporting: Most business applications need to deliver high-quality reports and visualizations to support users in making effective decisions. You not only need components for displaying those reports and visualizations but also a reporting tool for creating reports that work well with those components.

Your developers' productivity is going to be higher if the components and your backend tools "play well" together so that developers don't have to "stitch together" a solution.

Supporting the Inventory

Your ideal library will need to meet both the "good enough" and the "business critical"scenarios (and all the variations in between) while supporting your backend tools.

For the "good eough" scenario, "out of the box" (OOB) functionality matters: You want to be able to drop the component into your application, set one or two properties and move on. For the "business critical" scenario, you need the ability to customize the components to meet the needs of the more demanding UIs.

Component libraries support these requirements in two ways. Some libraries have a single component designed to handle both scenarios. That component's OOB functionality supports the first scenario (i.e., no matter how complex the "typical" scenario is, the developer can implement the "usual functionality" just by setting a few properties on the component). This, of course, provides much of the productivity gains that you want. In addition, the component provides multiple points of customization that let developers support more demanding/unusual scenarios—but, again, with far less developer effort than would be required by writing new code.

Looking at the documentation is helpful with components that implement this strategy: You'll want to make sure that the OOB functionality matches your typical scenario and that the component has a rich API that's easily exploited to meet more demanding scenarios.

The other strategy that component libraries implement is to have multiple, related components—each of which supports a different scenario. This strategy simplifies development because, even for demanding scenarios, there's a component whose OOB functionality meets the requirement.

With this strategy, you can look at the documentation to answer three key questions. The first two questions are "How easy is it to find all the related components?" and "Is it obvious what scenario each component supports?"

Once you've satisfied yourself on those two criteria, you need to ask the third question that addresses the "library benefits": Do the related components work the same way? That ensures that, as your UI evolves from simple to demanding (which happens a lot), you can replace a component with a related one without much effort.

It's also worth remembering that your users spend more time with everyone else's applications than with your application. Those other applications are setting your users' expectations. Where components don't work the way users expect, you're either going to incur coding costs to implement that expected behavior or training costs in getting users up to speed on "unexpected" behavior. The components' OOB behavior should meet all the "expected behavior" criteria for the applications in your inventory.

Does It Do the Job?

Next, you'll need to look at the detailed functionality embedded in your inventory.

Some of your requirements will be common to all business applications. Almost every business organization, for example, needs grids to display complex lists of data. Not surprisingly, then, most libraries include one or more data grids. You'll want to determine what interactivity features are available in those grids: Can users manipulate the grid's display, sort and filter data, expand rows to see related data, select records to edit or delete, and add new data from within the grid?

With data-oriented components, size also matters: Will you need to manipulate large datasets? If so, what are the limits for the data-related components in the library and what impact does that have on performance (assuming that performance matters to you)?

When discussing component libraries, you're usually focusing on UI components—you might think that security doesn't matter. That would be naïve. If you're building public-facing apps, for example, you'll want to know if there's a CAPTCHA component you can use. If your data-related components are going to be interacting with Web services and databases then, at the very least, you want to know how the components handle providing credentials and if they support securely transferring data.

Don't forget your reporting needs. That starts with creating UIs that include data visualization (e.g., dashboards)? If so, does the library include common dashboard components (graphs, sparklines, gauges, etc.) and does the library support the level of complexity you require?

If your UI goes beyond those visualizations to include reporting, recognize that you need compatible frontend components and backend tools. For example, do your reporting needs include a backend tool that supports creating reports that allow the user, in the frontend component, to drill down to the data they need? Do you need to go further and integrate the frontend component and backend tool to allow users to generate the reports they need "on the fly" from inside the application?

Your requirements might be quite simple and, if so, there's no point in paying for visualization components or reporting tools that support complex needs. The reverse is also true: If you intend to build sophisticated visualizations or interactive reports, you want a library that can deliver on both of those needs (you may, in fact, be slipping into that "specialized components" area discussed earlier—if so, you should go back and start there).

One more example: Do you need scheduling or other date-based functionality? If so, do the date-related components in the library meet your needs?

Obviously, this list of examples is potentially endless. This is why you need to develop your application inventory—it will let you focus on the criteria in this area that matter to you.

The Test Drive

You're now at the point where you should consider taking some sample components out for a test drive (most libraries provide a free trial period that will let you run a test).

In your test drive, pick a component from your inventory that's critical to your applications (probably a data grid) and some other component from your inventory that's, let's say, more obscure (e.g., an appointment manager or wizard builder). Make sure to include both OOB and non-OOB requirements so you can see what it's like to meet both scenarios.

The criteria you'll add to your list here begins, of course, with "Can the component do the job?" and "How well does it meet the requirements?" But also: "How long does it take a developer to get up to speed with the component?" That criteria assesses the developer API and how well that API adheres to current best practices in application development.

Library-Related Benefits

If you're getting a library, it's because you want the "library-related" benefits: Do the components play well together?

For example, in the user experience area, your users value consistency in their UIs more than almost anything else. As a result, having components that look and work the same across all your applications/platforms has real benefits.

A component library should also provide you with an easy way to change the appearance of all the components used in any user interface rather than having to adjust each component in a UI individually. Ideally, that should work the same way on every platform you support.

While it's certainly great if the component library comes with a set of predefined styles (again, ideally, that are cross-platform), the reality is that your organization will have a standard style that you'll want to apply. Does the library provide a tool for generating a style for your organization or a way of importing your organization's existing styling resources. Can you, for example, import your existing CSS sheets?

If you're using a UX design tool (e.g., Figma), it would be great if there was some integration between your design tool and the component library.

You should also expect components in a library to share a common approach so that developers can leverage what they learn from using one component when working with other components in the library. This consistency in design helps ensure that developers do get better and better at using a library's component over time. You'll want to pay special attention to this benefit when assessing open-source libraries because it's a harder goal to achieve in an open-source project due to the number of contributors.

Future Proofing

And, finally, you need to consider the "total cost of ownership": The ongoing costs a component library incurs over its life.

Focusing on this category, you're going to be looking at how important is to you for your components to keep current with evolving standards, technology and best practices. If you're in a highly regulated industry (e.g., finance or health care, among others) then you know that those regulations are constantly changing and you need components that keep up to date. Even if you're not in a highly regulated industry, you still need to care about changing security and accessibility requirements.

But even outside of those external forces, you'll need to consider whether you care about "staying current." If you're building customer-facing applications, for example, then you want UI components that follow the MAYA design principle (Most Advanced, Yet Acceptable) and, as a result, always look up to date without violating user expectations.

If you're building internal applications, you may be less concerned about MAYA (though obviously out-of-date UIs do send a message to users about how much you care about the applications your users have to interact with).

To assess these issues, you should look at the history of the library's evolution and the roadmap for the library's future (and it's a red flag if there *isn't* a roadmap). Part of looking at the roadmap and history is checking to see how well the actual history has matched the proposed roadmaps.

You'll want to pay particular attention to the library's maintenance history: When bugs are discovered, are they addressed in a timely fashion? If this is an open-source library, are you expected to provide solutions for bugs you find or does the existing community address them quickly? If you are expected to provide a solution, is this something you want to participate in?

Your earlier test drive will provide some additional insights here. What was the learning curve like for working with new components in the library? Is it the typical case: a gradual slope from easy for the OOB things and hard for more complicated requirements? Or is it like SQL learning curve: Anything easy is easy and everything else is really, really hard (well, until you hit the SQL "A-ha!" moment)? Or is it the ideal, flat "curve": Anything you need to do, other than the OOB stuff, is about as hard as anything else (and none of it is *very* hard).

Another ongoing cost will have surfaced during the test drive is how often your developers needed outside help. If they did (and they probably did), were there learning materials available from the library provider that met your developers' needs?

Does "support" mean there are multiple ways for developers to reach out for help and do some of them give developers access to the team members that built the components? Or does "support" just mean that there's a forum where you can post questions and hope for an answer from some other user? An active user community that can provide missing material is a good thing … but it probably shouldn't be the only thing.

The key question for any of the support mechanisms is how long it takes to resolve a question. In the long run—and you're hoping for a long run—lost developer hours spent waiting for support (or cycling through answers that don't work) can be the biggest cost in using a library. A test drive can let you assess the quality of support available to a library.

Shortening the List

That's a long set of criteria to consider but you'll only need some of these to create a checklist that's meaningful to you. While that may still be an imposing set of criteria, you may not need all of it—your first few criteria may get your list of libraries down to a single choice.

If, after you've applied all your criteria, there's still more than one choice on the list, pick the cheapest one. If all your remaining choices cost about the same, pick the library with "extra" components or the one that supports a platform you aren't yet targeting (because who knows that the future holds?). If there's still more than one choice on your list, let the developers who will have to live with the components make the choice—after all, every choice on the list is a good one.

But, regardless of which library you get, you'll know that you've got the right one for your organization.

If this paper struck a nerve, consider the Telerik and Kendo UI component libraries. These libraries are built specifically to help developers faced with complex UI requirements and to simplify the answer "How am I going to build that?" Each library is designed to give developers every component they need to build UI from the most basic to the most complex. Extras include design tools that eliminate most if not all CSS work, embedded reporting tools, professional support and frequent updates.

The Telerik and Kendo UI component libraries cover virtually all platforms—take a look at the most popular components for these platforms:

- Blazor: Learn about Telerik UI for Blazor's most popular components through the context of an expense management app.
- .NET MAUI: Read an overview of five most popular components in Telerik UI for .NET MAUI.
- Angular: Read an overview of five most popular components in Kendo UI for Angular.
- React: Read an overview of five most popular components in KendoReact.

The Telerik and Kendo UI component libraries are fully integrated with Telerik Reporting tools that not only embed reports views into your applications, they also can bring report design directly to your users. Read the guide.

Keep in mind, the content listed above covers some of Progress's most popular technology. Their portfolio covers virtually every platform. See the complete list and try them at telerik.com.

Learn more about and download trials of the Telerik and Kendo UI

About Progress

Progress (Nasdaq: PRGS) provides software that enables organizations to develop and deploy their mission-critical applications and experiences, as well as effectively manage their data platforms, cloud and IT infrastructure. As an experienced, trusted provider, we make the lives of technology professionals easier. Over 4 million developers and technologists at hundreds of thousands of enterprises depend on Progress. Learn more at www.progress.com

Worldwide Headquarters

Progress Software Corporation
15 Wayside Rd, Suite 400, Burlington, MA01803, USA
Tel: +1-800-477-6473

facebook.com/progresssw
twitter.com/progresssw
youtube.com/progresssw
linkedin.com/company/progress-software
progress_sw_

Progress®