9 Must Decisions in Desktop Application Development for Windows

General / November 20, 2018

Desktop application development dominated the software world for many years. However, with the rise of the internet, web applications took over with an incredible pace. When smartphones became popular, mobile applications came to be in huge demand, pushing desktop applications into third place.

While the previous statement might be true to startups, there is still an incredible amount of software development happening in desktop applications. Consider programs like **Excel**, **Adobe Photoshop** and the **Chrome browser**.

Besides similar existing monstrous applications, there are many good reasons to develop **new** desktop applications. Some of those are:

- The application doesn't have to be connected to the internet
- You can interact better with the user's PC. Web applications run in a sandbox environment and block almost all interactions.
- Desktop apps have better performance than web apps

- Running serious algorithms on the client side is possible but much harder with a web application.
- Utilizing Threads is much easier and more effective in a desktop application.
- Sometimes you don't care if the application will be Web or Desktop, but your team is more experienced with Desktop technologies

Developing for Desktop is great for a lot of reasons. You get to work with excellent mature technologies that stood the test of time. The debugging is as best as it comes. Arguably, desktop apps have less complexity and development is easier. In short, you're in luck!

Like with all software, there are many different ways to go when developing desktop applications. This includes different programming languages, frameworks, and architecture decisions. All of which will differ according to the individual needs of your product.

This article will show 9 of the most important things that need to be considered in advance or at the start of development. Considering this in early stages can save you rewriting code, doubling back in technology or choosing something that you will later regret but unable to replace.

1. Choose a UI Development Technology

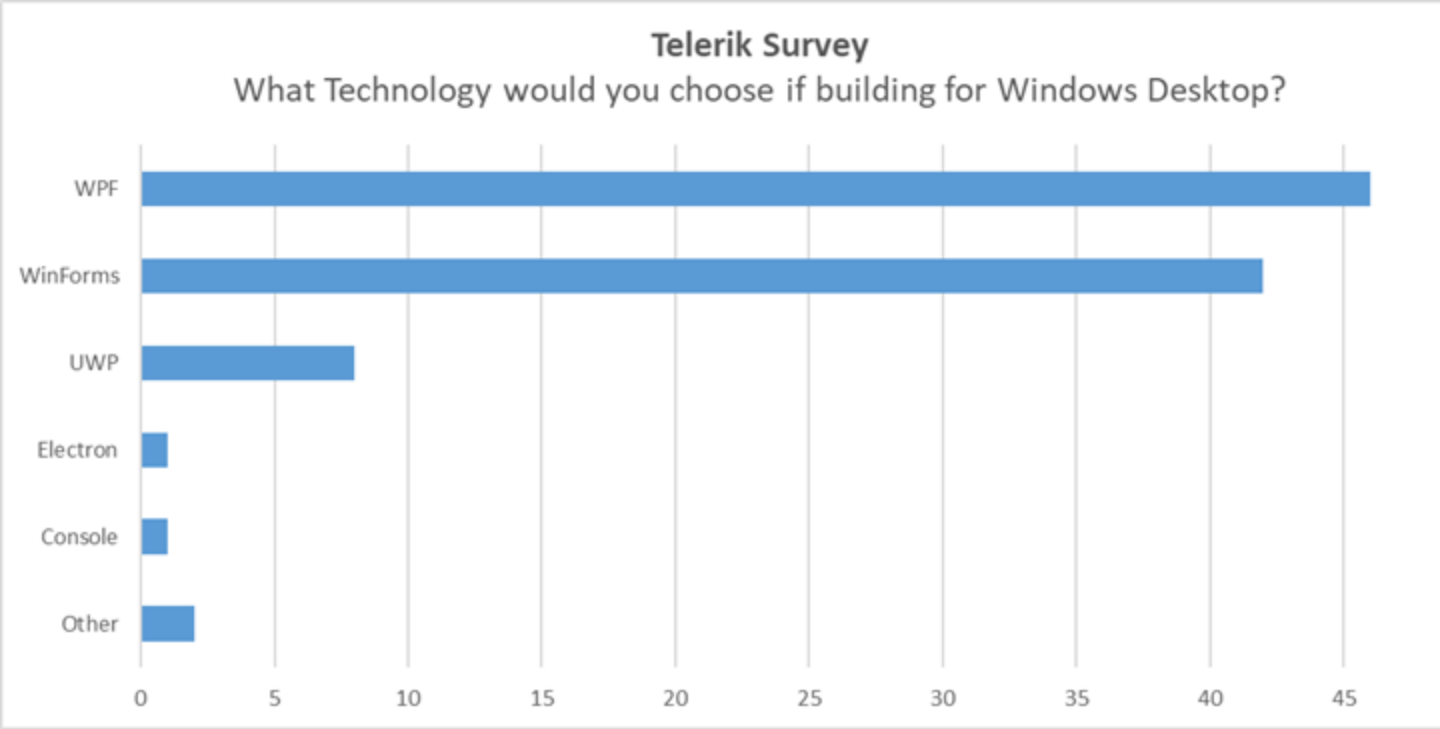
The first order of business is chossing your UI technology. There are many great UI frameworks for Windows, each with its own advantages and limitations. Most technologies will limit you to a certain programming language, which should be an important consideration according to your team's capabilities.

Here are the top technologies available today:

- UWP – Microsoft's newest Desktop Application technology. It's XAML based, like WPF, and you can write in C#, VB.Net, and C++ but most applications are written in C#. Key points on UWP:
  - The application works in a *Sandbox Environment*, so you are limited in your interaction with the PC.
  - Works only on Windows 10
  - The deployment is through Microsoft Store. This will make deployment and charging easier, but Microsoft will take a share of your profit (30% in fact).
  - UWP has a relatively steep learning curve
- WPF – A popular mature XAML based Microsoft technology. You can write in C# or VB.NET.
  - WPF has very powerful **Templating**, **Styling,** and **Binding** capabilities that are fitted for big applications.
  - WPF has a relatively steep learning curve.
  - Runs on any Windows OS.
  - A mature technology, available since 2006.
- WinForms – An older Microsoft technology, very popular before WPF. Unlike WPF and UWP, WinForms relies on Visual Studio Designer's drag and drop interface.

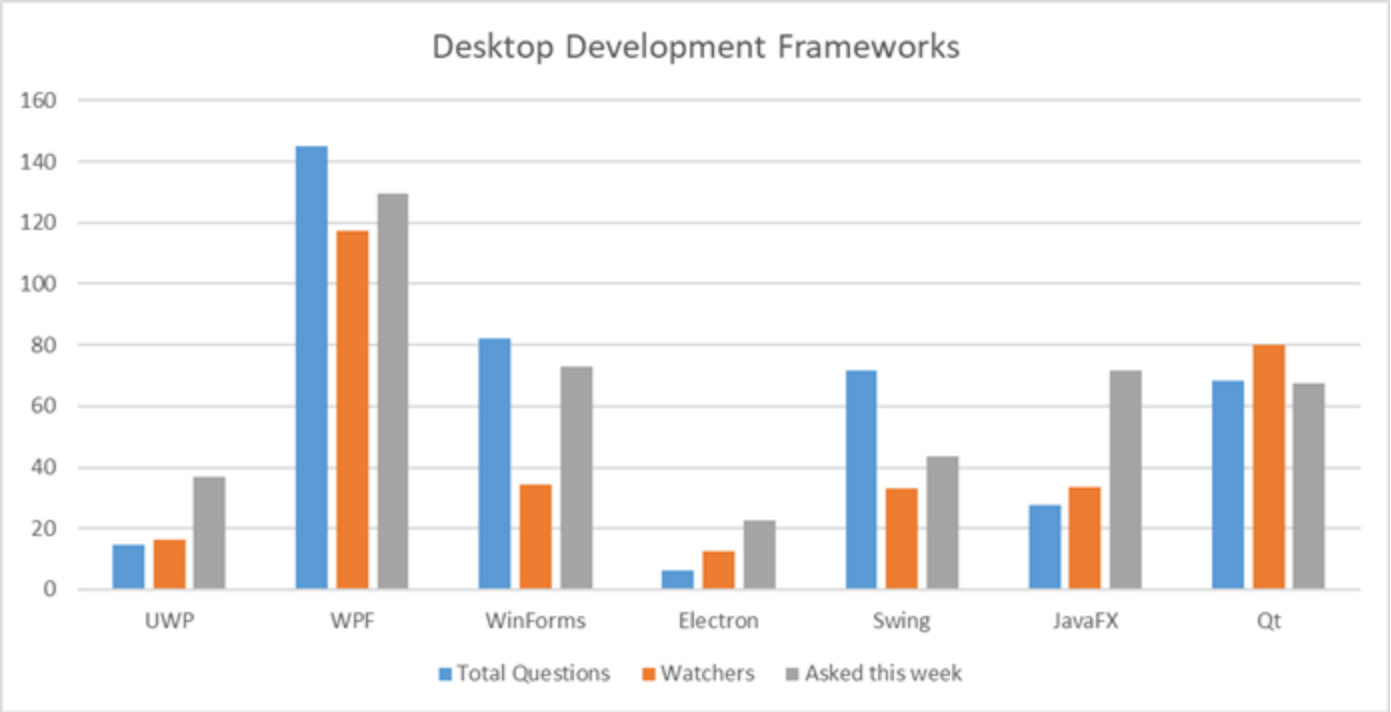
    - The drag and drop designer makes WinForms very productive for applications that don't require specific customization and Responsive UI.
    - Easy to learn
    - Works on all Windows versions
- Electron – A framework that allows developing Desktop apps with Web technologies (HTML/CSS/JavaScript). The magic behind Electron is that it uses **Node.Js** and **Chromium** to create a Web View in a desktop window. Electron gained some popularity for a while and there are great applications developed with it like **Slack**, **GitHub Desktop** and **Visual Studio Code**.
    - Electron is cross-platform
    - Has a big memory footprint, even for a simple 'Hello World' App
    - Heavy CPU consumption
    - Runs on Windows 7 and older
    - Interacting with the PC is much less capable than in other technologies. It's still possible though by using Electron's slim API for basic OS features or invoking .dll files directly with Node.js.
- JavaFX and Swing – Two Java UI frameworks from Oracle. Both are cross-platform. **JavaFX** is newer and encouraged by Oracle as a replacement for Swing, but it has relatively low adoption.
  Swing, on the other hand, is older and more widely used, but still less popular for Windows applications than WPF or WinForms. An example application written with Swing is JetBrain's **IntelliJ**.
    - Both are cross-platform
    - Swing and JavaFX are written in Java, so your end-user will have to install the JRE
- Qt – A cross-platform, C++ based UI framework. You can write the UI objects in code or use **QML**, which is a declarative language somewhat similar to JSON. An example of an application written with Qt is Maya.

To help you choose, you can look at Telerik's survey of 2016 where developers were asked: "What Technology would you choose if building for Windows Desktop?". The results are as follows:



According to this Survey, WPF and WinForms completely dominate the market. Also, it's pretty amazing that WinForms remains so very popular after all those years.

Telerik's survey might not represent the whole market though since Telerik customers are mostly in the .NET Controls space. It's also a bit outdated. So another way to check for popularity is a simple research in **StackOverflow**. I searched the **Tags** of all the technologies and compared them. Here's what I found:



WPF, for example, has 145K total questions, 23.5K watchers and 259 questions asked this week. **Electron**, on the other hand, has 6.1K total questions, 2.5K watchers and 45 questions asked this week. The exact data can be seen here.

You can argue that the amount of questions does not represent the framework's popularity and that a perfect product with intuitive API might have very few questions asked. But I think in reality the amount of questions is a very decent representation of a product's usage. Assuming that is the case, it's pretty clear from this data that **WPF** is the rock star, whereas **UWP** and **Electron** are the least popular. The only surprise for me was the relative popularity of **Qt**.

It makes a lot of sense to choose a popular technology. It will have more experienced developers available for hire, more tools, more documentation, and more 3rd party libraries. In addition to that, there's a better chance that it got to be popular due to its merits.

I am personally a big fan of WPF. It has a lot of advantages including the best **Binding** mechanism and the best **Templating** infrastructure that I ever worked it.

Having said that, there are many points of consideration when choosing a technology:

- When building cross-platform the only options are Electron, Qt, Swing, and JavaFX
- If you have a web product, consider Electron to reuse your available code
- If your team is web-oriented, consider Electron
- If your team is Java-oriented, Swing or JavaFX make a bit more sense
- WPF and UWP have a steeper learning curve than WinForms (but it's worth it if the application is big enough)
- UWP, WPF, and WinForms have amazing Control suites from companies like Telerik

- In Electron you can take advantage of the huge amount of web libraries and controls available
- While C++ is not the most productive language when compared to C# or Java, it makes sense to use Qt when your application is heavily interacting with a C++ native layer. Note that It's also possible to use .NET with a C++ layer using PInvoke or C++/CLI, but it requires more effort.

2. Choose a Deployment Strategy

How do you plan to deliver your software to your customers? This should be a business decision before considering technology options. Consider what are your limitations and what is your optimal customer experience.

For example, the **Chrome** browser is a Desktop Application with automatic updates that the customer doesn't even know about. That might be great for a **Consumer App**, but **Enterprise App** clients can be very picky about updates. They might want to be explicitly informed about the upcoming update and its release notes.

Some customers, like military device applications, might not have internet access at all and the update will have to be performed with a Flash Drive installation.

My point is that deployment is foremost a business decision and only then technological. Now that that's clear, let's see our various options:

- One way to go is Windows Store. It takes care of publishing and updates for you. A very convenient option if it suits your business needs and you're willing to live with the limitations (Windows 8.x+ only, sandbox environment, etc.).
  initially, only UWP Apps were allowed to be published in the Windows Store. However, Microsoft released the Desktop App Converter, which can wrap a WinForms and WPF application into a UWP application and publish to the Store.
- ClickOnce is Microsoft's solution to deploying your WPF app (but not UWP). ClickOnce takes care of packaging your App, Installing it and Updating it.
  There is some bad reputation around ClickOnce in the community and I would be very careful about choosing this technology.
- Squirrel – Another Installer and Update framework, like ClickOnce. In fact, its slogan is **Squirrel: It's like ClickOnce but Works™**
  I heard good things about it and it's worth to research and see if it suits you.
- Chocolatey is an interesting solution to distribute your app and easily publish updates. It requires the user to install Chocolatey on his PC and then use the command line to install and update your app. Doesn't sound user-friendly but on further investigation, it actually seems pretty awesome. More suited for developer customers.
- **The custom solution:** Use an **Installer** and develop the update mechanism yourself. From my experience, this is the preferred solution for many companies.
  The idea is to publish your product version install files to a known network location, and the Desktop Application will endlessly query that location for new updates.

Of course, you will need to handle a bunch of issues yourself like **Manifest files**, **Certificate verification** and so on.

3. Choose an Installer

An Installer is part of the deployment process, but I set it apart since it's such a big piece.

An Installer's job is to *package* your application into an installation program. This might be a *setup.exe* file or an *.msi* file. On deploy, the user or your application will run the installation program which will copy your packaged application to the current PC and do a bunch of additional jobs like: Write to Registry, Add Start-Menu shortcuts and associate files to your program.

Deployment tools like ClickOnce and Squirrel already include an Installer of their own.

There are some great installers at your disposal:

- InstallShield – Claims to be the industry standard installer for Windows applications. Well, it sort of is. It's very feature rich and always up to date with the latest technologies. It can create MSI, EXE, and UWP app packages installers. You can work with a development studio to choose all the tasks the installer should do. Also, InstallShield has its own scripting language called **InstallScript** to write custom jobs. It's pretty pricey, starting with about 700$ for the most basic plan.
- Inno Setup is a popular free installer, I used it and was very happy with it. You basically create a text file (.iss) file which contains your installer's settings and scripts.
  Scripts are written in **Pascal** language.
  InnoSetup has good documentation and a good-sized community. It produces only EXE files though, not MSI. On an update, InnoSetup will uninstall the previous version and install the new one.
- Wix is another popular free installer. It has a steeper learning curve than InstallShield and Inno Setup, but it can produce MSI files which can be a big advantage.

4. Setting up Continuous Integration and Deployment (CI/CD)

Setting up a CI/CD pipeline is very important to a healthy development process.

The idea behind those concepts is to create an automated workflow for your development. An automated flow is triggered on demand or after a code check-in. It can do the following:

- Pull latest code from repository
- Compile your code and check for errors
- Run Unit Tests
- Create an Installation package
- Deploy your application if needed

The above bullet points are just an example of course. Each project will have a different automation flow, and possibly several of them.

There are great automation tools on the market like **Jenkins** and **Azure DevOps** (previously known as **VSTS**). I wrote extensively on the CI/CD process in a previous article: Creating an ecosystem with Continuous Integration, Installer, and Deployments for WPF/UWP application.

5. Logging and Troubleshooting

Logging in a Desktop Application is very different from logging on a server since you don't have access to your customer's computer.

There are several standard logging and troubleshooting solutions you can (and should) implement in your Desktop Application:

- Standard logging to file. There are several logging libraries available, most known being **Log4net** and **NLog**.
  Logging to a file is very effective because it's easy to find the file and retrieve the logs when needed. You will probably have to ask your customers to retrieve logs and send them when they have a problem, so it's important to make this as easy as possible.
- Logging directly to Windows Event Logs is another fast and reliable way to log, though It will be a bit harder to retrieve those logs. Both **Log4net** and **NLog** can write to Windows Event Logs.
- **Reporting a Problem** mechanism is a great way to monitor your Desktop Applications. After recognizing an Exception or Crash you can send your log files and exception details to some designated server. Some applications choose to ask the user's permission and others just do it silently.
- **Saving a Dump** – Another standard procedure is to automatically save your application's Dump on Crash (or Hang). Similar to a Picture being worth a thousand words, a **Dump** is worth a thousand log files.
  You can attach your dumps to the **Report a Problem** mechanism.

I suggest reading Debugging a .NET program after crash (Post-morterm) debugging to better understand what pieces you can use after your program crashed and how to use them for debugging.

6. Theming your application

Having much experience in this matter, I would suggest creating your application with Theming considered in advance.

There are two common scenarios where you would need Theming:

1. When you are building a B2B application, your customers might want different branding for the application. So if you're building a cashier app for McDonald's, they might want the colors to be the McDonald's Yellow and Red.
2. For a consumer app, you might want to give your users a choice for a custom look and feel. This might be different colors, different control styles and so on.

If you do need Theming according to your business needs, you should architect your application accordingly. For WPF, I suggest reading my article WPF complete guide to Themes and Skins.

Note that some Control Suites (see next) include support for popular themes like Material Design.

7. Purchasing Controls

Depending on your applications, you might want to consider purchasing a Control suite.

Companies like **Telerik**, **DevExpress**, **Xceed**, **SyncFusion**, and **Infragistics** offer sets of ready controls with advanced functionality, that you won't have to develop yourself. These include DataGrids, Charts, Carousels, TreeViews and much more.

It might make sense for you to buy a set of such controls, since developing those yourself can be difficult and time-consuming. Especially if you want some of the more advanced features.

When considering buying 3rd party controls, I take into account these factors:

- How customized should your controls be? If your product team needs really specific controls, then you might as well make them yourself instead of buying 3rd party controls and modifying them.
- How experienced is your team? If they are not experienced, they won't be able to develop similar control themselves (in a reasonable timeline and effort) and you don't really have a choice.
- Price. Those control suites are pretty pricey, costing as much as a $1000 USD per developer license. Estimate how much money it will take to develop the controls yourself instead of buying them. Is it worth the price?
  There are some free suites as well. For example, Xceed offers a free toolkit.
- How much time for development do you have? Buying 3rd party controls will save a lot of time you can use for developing the rest of your app.

8. Choose your MV* framework(s)

According to your chosen UI technology, there's usually a recommended design pattern to use. In WPF it's MVVM. In WinForms it's MVP. There are excellent libraries that implement those design pattern and it's recommended to use one of them.

The most popular frameworks for **WPF** and **UWP** are MVVM light, Prism, and Caliburn.Micro. If you're building a lightweight application, **MVVM light** provides basic MVVM features. **Prism** is a monster packed with features, so it's more suited for large enterprise applications. **Caliburn.Micro** is convention-based and feels a bit like black voodoo magic, so it's a matter of personal taste. I didn't work with it personally but heard good things.

WinForms, on the other hand, is more straightforward, so there aren't any widely used frameworks.

Electron applications are web technology based and you can use **Angular**, **React** or **Vue.js** like in a regular web application.

Most of those frameworks are great and it's a matter of personal taste and your team's experience. For example, if your team is already experienced **Caliburn.Micro** and loves it, then there's no reason to change.

9. Licensing

If you don't publish with the **Windows Store** and your product is paid, then you need some sort of licensing system.

Basically, there are 4 types of licensing, which you can choose from or even combine:

- **Product key** – Your customer pays you and receives a product key string. That product key is entered within the application, validated, and everything works.
  Sounds easy, but there's a world of complexity here. Some key points are:
  - Do you validate the license key locally or on a network server?
  - Is the license valid for some time or forever?
  - How do you prevent multiple customers from using the same product key?
- **Login** – A more modern licensing system is to use **Login** of your use from the application. An example of this license is in recent Visual Studio versions, where you need to log in with your Microsoft Account.
  This is even less simple than the product key solution.
  - You'll need to create some sort of Identity Provider Server.
  - It's best to use standard Identification protocol like **OAuth**.
  - You can implement your own user database or allow to log in with an identity provider like **Google**, **Facebook** or **Microsoft**. A couple of ways to integrate with 3rd party identity providers is with Auth0 or IdentityServer.
  - How do you prevent multiple customers from using the same account?
- **Licensing Dongle** – A third, less common licensing method is with a **USB License Dongle**. This means you will give your customers a physical USB device that will act as a license.
  This kind of license method is suited for enterprise applications that require extra security. Perhaps in the government or for security organizations.
  The advantage is that the dongle vendor will provide you with a complete software

solution that you can integrate into your application. Usually much simpler than building your own licensing system.

- **Hardware License Key** – If you're privileged enough to sell the hardware along with the software, you have the option to burn a unique identifier into the hardware itself (product key or MAC address).
  This makes your job easier since there's no need to enforce multiple customers not to use the same product key.

Summary

Desktop Application Development for Windows is full of complexity like any software solution. We covered some of the most important decisions in an application's architecture but there are other considerations like **Localization**, **Testing**, **User Settings**, **Database** and so on.

I love Desktop Applications because I think the challenges are more interesting than in a typical web development. In a web application, I find myself dealing with Databases, ORM frameworks, Identity, and Authorization. In Desktop Applications there are usually more logical, algorithmic type of problems, multi-threading issues, and a lot of UI work (which I enjoy).

Thank you for reading and subscribe to the blog for more interesting in-depth articles on desktop application development and programming in general.

Share: