Resolute

Best Practices & Considerations When Modernizing Progress Telerik UI & Kendo UI Powered Apps

Whitepaper

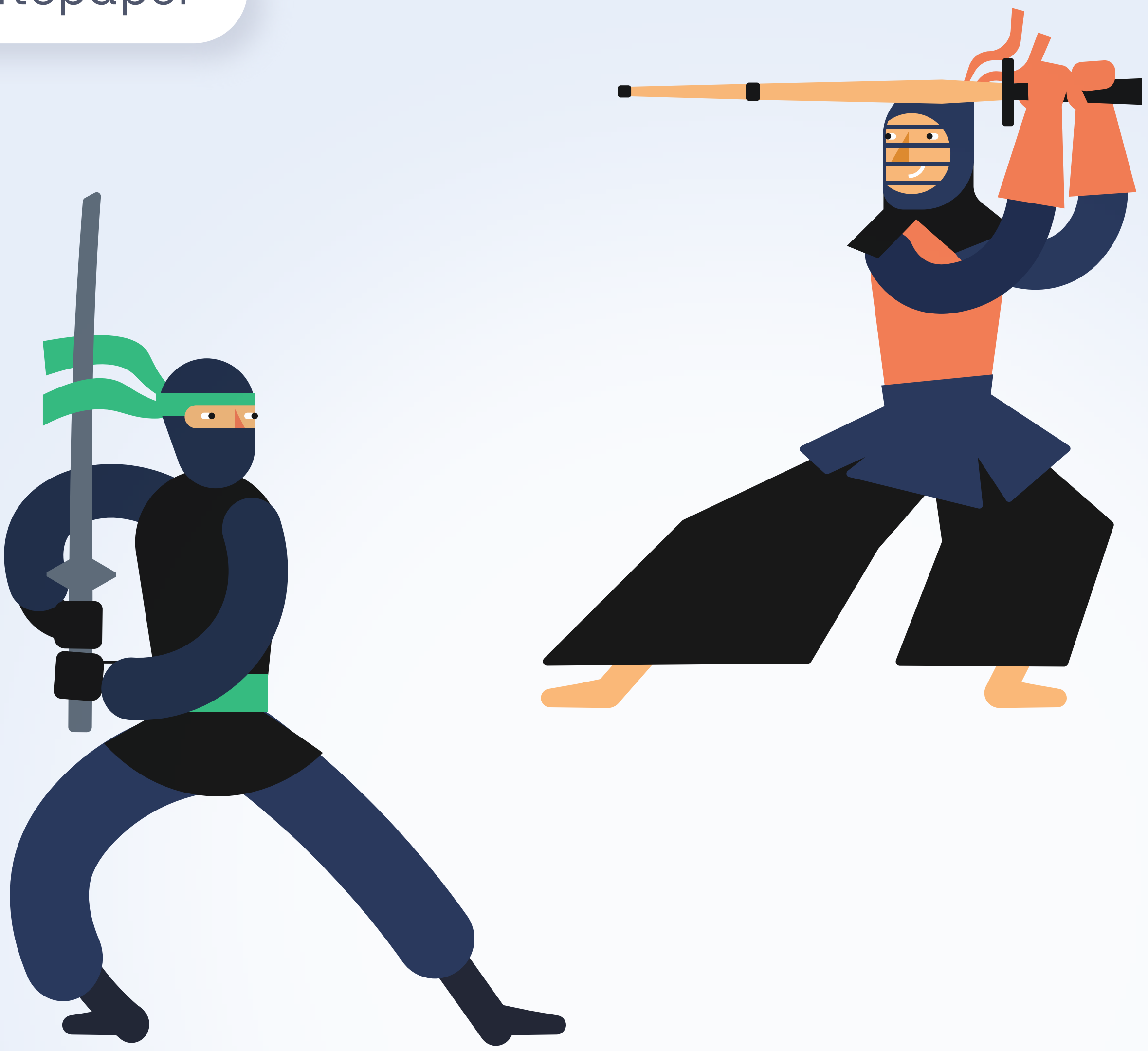


Table of contents

Introduction

Your legacy systems are **critical for your business operations.** Changing how they work often seems like a risky proposition. But at the same time, you can't help but notice that keeping them running grows more time-consuming and expensive each year. It's time for a change.

But where do you start?

This paper will look at **modernizing a Telerik UI-powered or Kendo UI-powered applications.** We'll look at application design, architectural issues, platform considerations, client technologies, and the other salient issues this project entails.

A Brief History of Telerik and Kendo UI

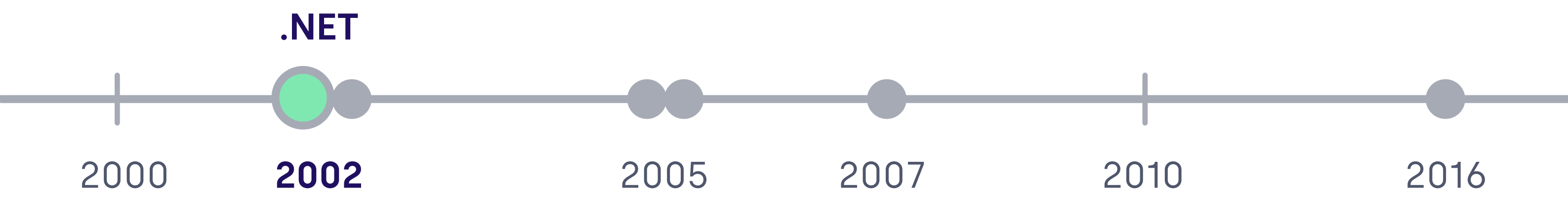
Telerik UI and its JavaScript offshoot Kendo UI grew and evolved in parallel with Microsoft .NET web offerings.

Telerik UI & ASP.NET

Initial Release

After two years of previews and betas, Microsoft released ASP.NET as part of the newly unveiled .NET component stack in 2002. ASP.NET integrated Active Server Pages, Microsoft's server-side scripting language, into the .NET Common Language Infrastructure (CLI).

Adding the .NET CLI to Active Server Pages gave web developers access to Windows dynamically-linked libraries (DLLs) with complete object-oriented language support, as well as advanced features like exception handling and type safety.

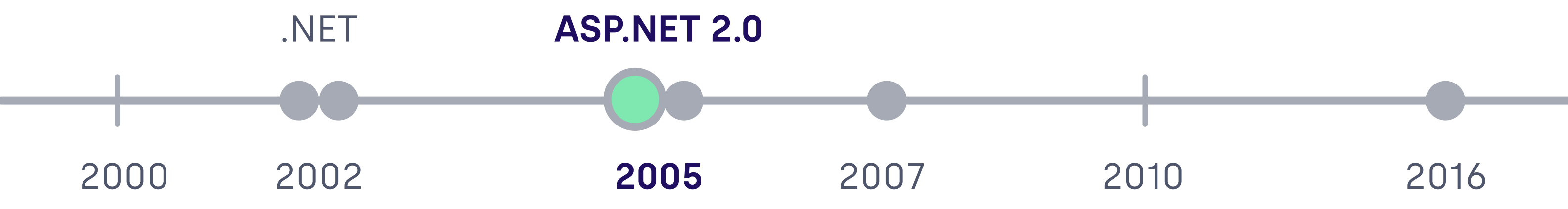


ASP.NET 2.0: “Project Whidbey”

While the introduction of ASP.NET was a landmark for enterprise web capabilities, the next major release was a milestone in developer productivity.

In November 2005, **ASP.NET 2.0 and Visual Studio 2005** gave web developers a broad set of new features, including:

- Declarative access to SQL data stores
- A generous assortment of new controls, including lists, file uploads, and mulitviews
- Enhanced localization features
- Object-relational mapping
- Advanced template support with Master Pages
- Improved page loading times via pre-compilation

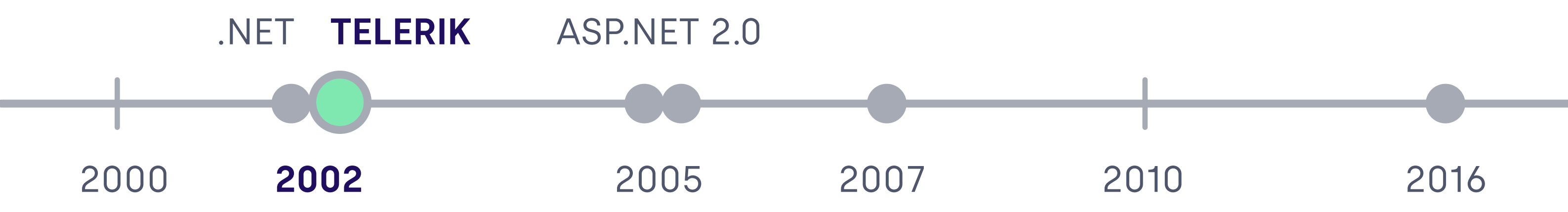


Telerik Sitefinity & the Genesis of Telerik UI

While Microsoft was working toward ASP.NET 2.0, independent software vendors embraced the platform with tools and libraries that supported building enterprise web applications.

Telerik, a small vendor founded in 2002, started as a .NET development tool company. In 2005, they expanded into the ASP.NET space with Sitefinity, a modular content management system (CMS). The software shipped with custom controls, a complete API, and built-in content management workflows.

Sitefinity's popularity and the success of its custom controls laid the foundation for Telerik's UI component libraries. By 2007, Telerik had released several versions of UI for ASP.NET, including RadAjax, its own AJAX engine for ASP pages.

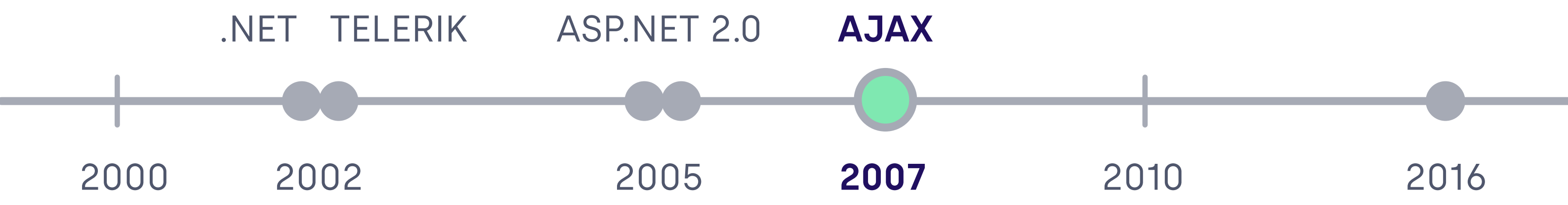


ASP.NET AJAX & Telerik UI AJAX

In January 2007, Microsoft introduced ASP.NET AJAX. This library integrated with ASP.NET 2.0 and made it easy for developers familiar with the .NET paradigm to build interactive AJAX interfaces. The client side libraries weren't exclusive to .NET and worked with other back-end systems.

In April of that same year, Telerik released UI for ASP.NET AJAX, code named **"Prometheus"**, adding AJAX to the UI framework.

This framework replaced RadAjax, which had merely layered AJAX interactivity over the existing .NET framework with controls implemented with Microsoft's new integrated support for AJAX.



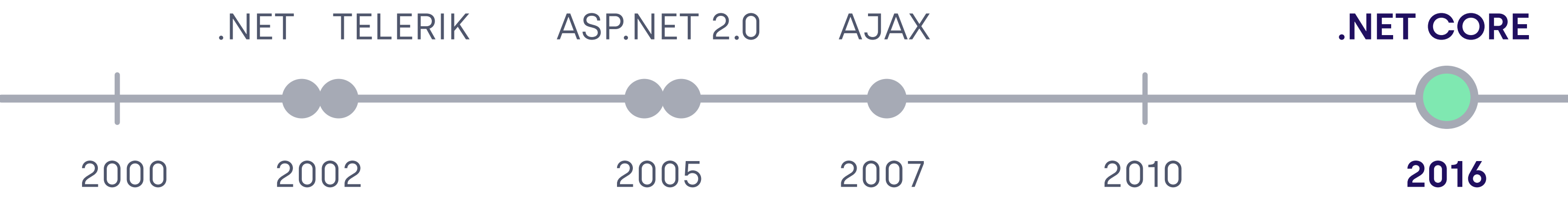
Telerik UI & .NET Core

In 2016, ASP.NET was transitioned to .NET Core. This re-branding reflected a rewrite combining ASP.NET MVC and ASP.NET Web APIs into a single model, as well as a move to open-source.

The new model introduced several **important features** for improving developer productivity:

- Modular packages distributed via NuGet
- Cross platform support for MacOS, Linux, and Windows
- Dependency injection
- Support for more than one library version on the same server

Telerik issued the first release of UI for ASP.NET Core, with a port of UI for ASP.NET MVC. Subsequent releases included support for JQuery, AngularJS, and better integration with Kendo UI via NuGet packages.



Kendo UI and the Shift Toward JavaScript

Kendo UI started as a set of **JQuery and AngularJS controls**, with DataViz being a popular offering for building charts for scientific and market applications. But as web development and .NET evolved toward more JavaScript-centric approaches, so did Telerik's offerings. Kendo UI grew from a set of controls to a component library with a compelling offering: **a consistent interface for the four major application frameworks (Angular, React, Vue, and JQuery).**

02

Planning an Application Upgrade

Planning an application upgrade includes a wide variety of concerns. But like any software design, you can avoid overwhelm and confusion by breaking the project down into separate areas of concern:

- Architecture
- Platform
- Client Technologies
- Design
- Business Logic
- Testing

Architectural Changes

Upgrading from legacy web technologies includes changing how clients and servers communicate, as well as how your back-end processes messages. Rather than re-architect these changes as your teams code, it's best to look at the new technologies and plan ahead.

Request Processing Models

Node.js and .NET use different threading models to manage client requests. These models are an important consideration when considering a port from .NET to Node.js.

.NET

.NET support both synchronous and asynchronous request processing, but its async model is different from Node.js.

With **Synchronous Request Handling**, requests are processed by a single thread. If the request requires a blocking operation, the thread waits. The threads come from a statically-allocated pool, so if too many requests are blocking, the application risks starvation.

Asynchronous Pages, introduced in ASP.NET 2.0, allow a developer to designate operations within as asynchronous. When they're encountered, they are completed on a new thread, and the request thread returns to the pool. When the asynchronous operation completes, another request thread is pulled from the pool to complete the request. **Asynchronous HTTP Handlers** are a similar construct that operate with the same threading model.

The .NET framework also has asynchronous support for HTTP modules and services.

Node.js

Node.js servers use a single event loop to dispatch requests and collect responses via callbacks. The worker threads that process the requests are also non-blocking.

This asynchronous model makes Node.js a highly responsive and scalable system, but it also means that any synchronous, compute-intensive, or time-consuming operations need to be off-loaded to an external service, or they will cause serious performance problems.

The callback model in Node is different from .NET in that the developer has more control—and more responsibility—over how asynchronous tasks complete and return their results.

Design Considerations

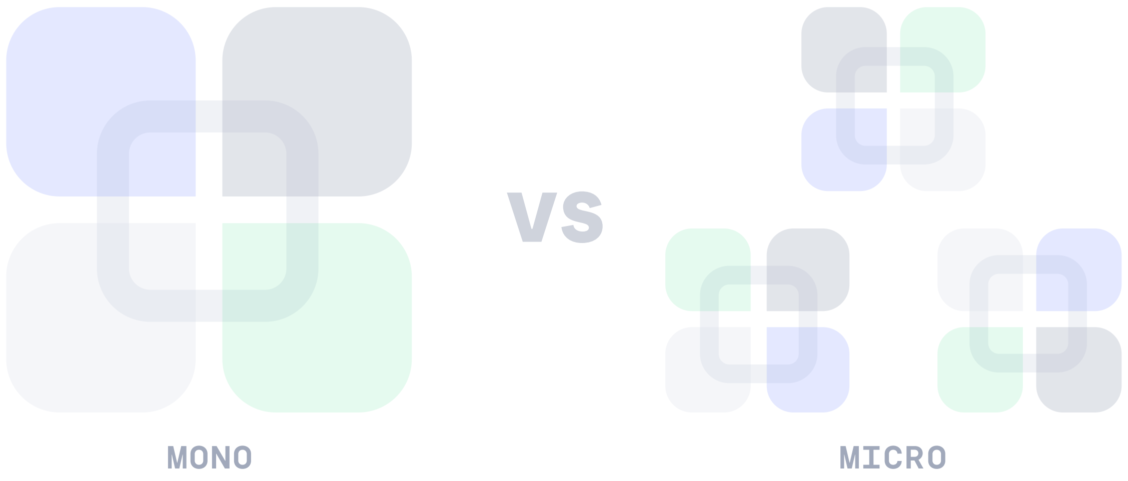
Because of Microsoft's mixed threading model, you'll need to carefully inspect your .NET code, identify synchronous operations, and either redesign them to be non-blocking, or find a home for them in a decoupled system, such as a microservice that sits behind the back-end service.

Microservices vs. Monolithic Services

A review of the system's overall architecture goes hand-in-hand with a review of request processing models and, as we'll see below, client-server communications.

Many legacy .NET applications are designed with a monolithic architecture. They're built as an integrated unit that runs on a small number of servers (often a single server or cluster) and is built from a single codebase. **This architecture has its strengths and weaknesses, and this paper is about considerations when modernizing an application, not advocating for one design paradigm over another.** Reviewing the architecture and evaluating how it fits in with the new components and protocols is a necessary step.

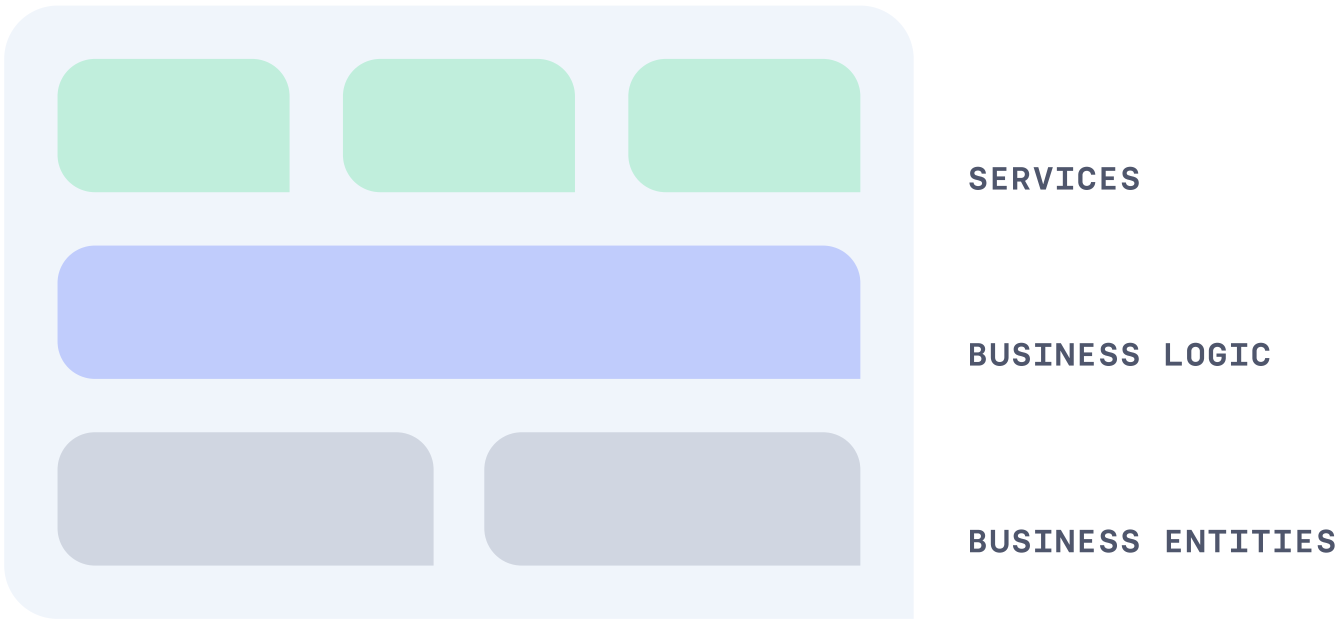
Breaking down the monolith is worth considering, especially if the modernization project involves redesigning request handling as we discussed above.



Monoliths

The obvious alternative to a monolithic architecture is one based around microservices, but it's not the only one, and it might not be the best. Another option is the microlith.

A microlith decomposes the web service into multiple services, **but it maintains a single connection to clients**.

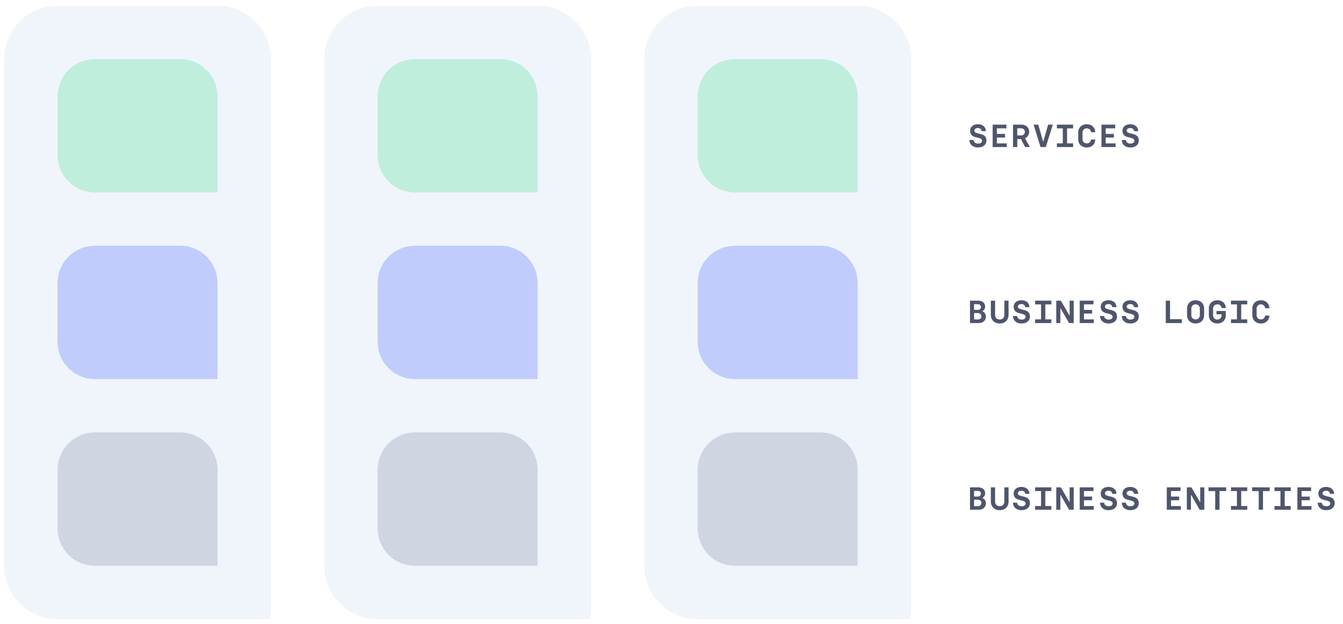


So a monolithic service that managed authentication and access to a relational database and to market data could be broken down into three services that sit behind a user session manager that has a basic understanding of business logic and knows how to orchestrate messaging between the individual services and clients.

This design establishes the same separation of concerns and flexibility as a microservice architecture, but it represents a more incremental change than a complete migration to microservices.

Microservices

Microservices go a step further. You break the monolith down into discrete services, and the client application communicates with each one. **Each service has its own message formats, business logic, and codebase.**



This loose coupling has many advantages, such as making it easier to design, code, test, and deploy changes into one part of the system without affecting others. But microservices can introduce complexity and even chaos into a system if there is no coordination between them.

It makes sense to look at how you'll structure communications between clients and servers before selecting an architecture.

Client Protocols

Microservices, RESTful APIs, and GraphQL are often associated with each other. **But even if you don't opt to shift to a microservice architecture, you need to modernize your client protocols to optimize and take advantage of the latest application tools.** If nothing else, moving away from SOAP's XML to JSON will save your application bandwidth and processing power.

But these REST and GraphQL represent more than just a way to format requests and responses between parties. They define query types and message semantics and manage the state of clients.

RESTful APIs

RESTful APIs have become the common dialect for many internet services. So much so that REST's wide adoption has become one of its primary strengths, if you build applications and tools that can speak REST, it's easier to integrate third-party services and libraries. It's also easier to export an API to clients and partners.

REST has a few important design principles:

- **Decoupling:** Servers and clients are independent of each other. Servers expose Uniform Resource Identifiers (URIs) that define how data is requested and received.
- **Stateless:** REST is short for Representational State Transfer. Requests and responses are discrete events.
- **Consistent Interface:** There should be one and only one URI for a piece of information in an API, and the API request for a resource type should always look the same

Decoupling is an important concept in RESTful APIs, and where it often differs from legacy applications. Clients call the API and receive a result that reflects the current state of a URI. That cannot make any assumptions about "who" they are speaking to. This separation makes it possible to layer a RESTful API over a legacy system.

GraphQL

GraphQL is a declarative query language that allows a client to formulate ad hoc requests from multiple services. The client can populate the queries with just the fields they need and omit the ones they don't. This simplifies data retrieval for the client, since they only need to speak with a single server, and they don't need to filter unnecessary information.

GraphQL avoids coupling between client and server with schemas that define what resources look like. The schemas make it possible for clients to see how resources are formed and generate queries for them.

Which One?

Which protocol is best depends on the data your application works with, and complete comparison of REST and graphQL is beyond the scope of this paper. But we can look at a quick example.

Imagine an application for managing music that has a page for displaying and editing albums. The page is a table with no artwork; it's only designed to update information about the tracks and artists.

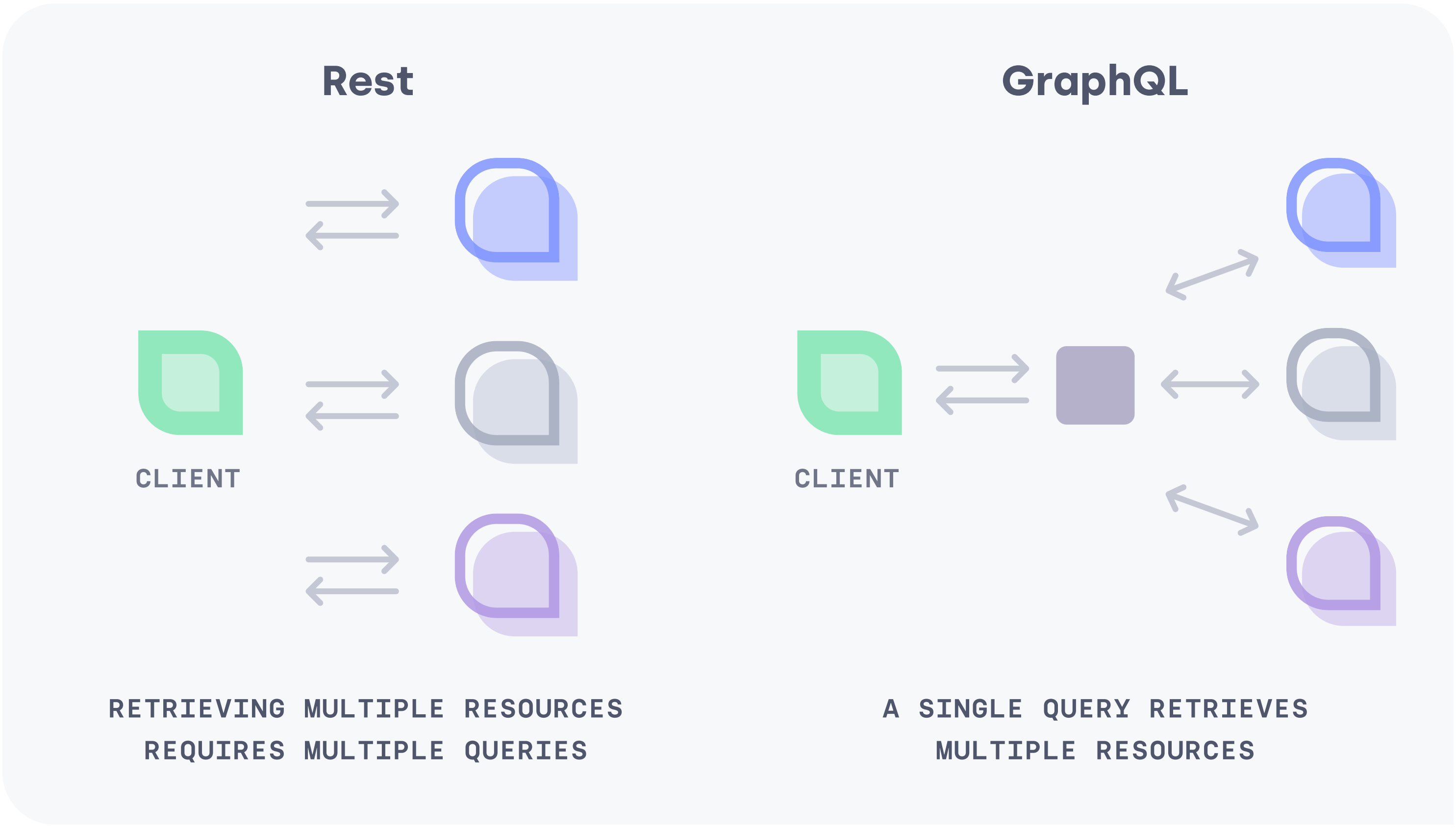
- ✓ An album has a title, a release date, a list of tracks, a list of images, and a list of artists
- ✓ Each track has a list of artists
- ✓ Each artist has a name

A RESTFUL APPLICATION WOULD:

- → Request the albums
- → Discard or ignore the dates and image URIs
- → Build a list of artists
- → Request the artists
- → Collate the albums, tracks, and artists into the table

A GRAPHQL APPLICATION WOULD:

- → Get the data schema
- → Build a query for the albums that specifies it wants artists, tracks, and artists for each track, but omit the rest of the album data
- → Use the results to build the table


Rest
GraphQL
CLIENT
CLIENT
RETRIEVING MULTIPLE RESOURCES REQUIRES MULTIPLE QUERIES
A SINGLE QUERY RETRIEVES MULTIPLE RESOURCES

On the surface, GraphQL looks more efficient, especially if you're designing or implementing the client. It makes a single request and displays the results. But the work of collating the data didn't go away—it was pushed onto the server. Depending on your application's needs, this may or may not be the best approach.

This example brings up another question. **What's more important, network bandwidth or processing power?** For REST, the client made two requests and then merged the lists while discarding some extra fields. GraphQL made a single request. How big was GraphQL's response? How many duplicated artist names did it contain? REST doesn't have a built-in mechanism for filtering unwanted fields. GraphQL doesn't have any built-in mechanism for compressing duplicate data.

Platform Considerations

Modernizing an application is an opportunity to examine every aspect of how, where, and why it works. There are benefits to staying with your current platform, but there may be reasons to move to a new one, too.

Linux vs. Windows

Node.js and .NET are cross-platform technologies. Most companies opt for either Windows or Linux, with Linux being a more common choice for Node.js.

When evaluating the two operating systems, consider the following:

- **Licensing Costs:** Windows always comes with a licensing fee, while you can run Linux for “free,” or with a support contract from one of the major vendors.
- **Support Staff:** if you switch from one operating system to another, you’ll need to train, augment, or replace your current support staff.
- **Infrastructure Support:** Node.js and .NET are cross platform, but what about the rest of your infrastructure? Will you need to support multiple operating systems?
- **Cloud vs. On-Premises:** Will you run the application in the cloud? Have you chosen a provider? Which operating system do they support best, and what are the costs?

Cloud vs. On-Premises

Many companies elect to move to the cloud because they expect to save on costs, and many are disappointed when that doesn't turn out to be the case.

Costs

Cloud computing involves considerably lower capital expenditure (CAPEX) than on-premises, but it comes with higher monthly costs. You may be able to control those costs by strategically scaling your resources up and down based on demand, but only if your application supports this strategy.

Security

Cloud security is often cited as a risk. When you extend or move your applications to a cloud provider, you are essentially dependent on their security measures. But are your security practices better than theirs? You may be improving your security with a move to the cloud.

Staffing

Eliminating or reducing your on-premises hardware means you need less hardware support staff. But maintaining cloud infrastructure requires a new skill set in high demand.

Cloud Options

Moving to the cloud can mean many things.
Here are a few common options:

1. **Dedicated Servers:** A dedicated cloud server is "someone else's computer." You're renting a dedicated resource located in a cloud data center, and your application has exclusive access to the system's memory, CPU, and disk.

2. **Virtual Private Servers (VPS):** A virtual machine running on a shared server. In this scenario, you share a host with one or more clients. The system's hypervisor limits the resources your VPS can access, and you're not protected from a misbehaving application running on the same shared host.

3. **Cloud Instances:** A cloud instance is a virtual system similar to a VPS but better isolated from other clients. You can use them for long-lived purposes, like a dedicated server, or scale them up and down based on demands. Cloud instances are generally based on the active time rather than billed monthly like a dedicated server or VPS.

4. **Serverless:** A serverless application isn't tied to a system. It runs on demand and then exits. These applications have significant limitations with regard to how long they run and how they manage state, but they can be very cost-efficient for some applications.

Selecting the Right Client Technologies

Which front-end libraries and toolkits are best suited for your project? Several options exist even if you decide to stay with Microsoft and Telerik.

Incremental Change?
Or Rewrite From the Ground Up?

How will you approach modernizing your client applications? Will you take an incremental approach, or start from scratch?

This is just as much of a business decision as it is a technological one. Developers are usually ready to start over and fix the mistakes of the past. But how long will it take to reproduce all the existing functionality? What happens to bug fixes and new feature requests on the legacy platform in the meantime? These are age-old issues, and the solutions depend on individual circumstances.

An incremental approach skirts many, but not all, of those issues. It's easier to carve out a portion of an app for a code freeze while building the replacement, and it makes it easier to demonstrate steady progress to management. But incremental change means a prolonged period where you need to support two or more different architectures.

Selecting the Right Frameworks and Tools

Cloud computing involves considerably lower capital expenditure (CAPEX) than on-premises, but it comes with higher monthly costs. You may be able to control those costs by strategically scaling your resources up and down based on demand, but only if your application supports this strategy.

ASP.NET MVC

Blazor

Blazor is a single page app framework that uses C# instead of JavaScript. If your developers are familiar with C# and see advantages to working with the same language in the client as the back end, Blazor is an attractive option.

Razor Pages uses the Model, View, ViewModel design pattern for pages. It's a popular model for mobile applications and JavaScript libraries like Knockout.js. This programming model often leads to simpler code that is easier to understand and test. Razor is an excellent candidate for an incremental migration of a legacy ASP.NET application.

Razor Pages

Angular, React and VueJS

Finally, there is the option to move away from .NET technologies on the client side. Angular, React, and Vue enjoy wide community support, a variety of components, and support from third-parties like Progress Telerik.

Design Considerations

Modernizing an application can be a daunting task, and the design plays a critical role in ensuring its success.

Documenting and Preserving Existing Functionality

Regardless of whether you opt for an incremental upgrade or a complete rewrite, **you need to document how your application works and identify the features and behaviours you'll retain in the next version.**

Some features will be required because of legal or contractual obligations, others will be required because clients expect them, while some features may be obsolete or unused. Gathering this information may be the most difficult and gruelling step in the upgrade process, but if you get it right, then you'll have a clear and correct road map for the project. If you don't, you'll be setting your team up for unpleasant surprises.

Sources for Documentation

There are several primary sources for the documentation you need to plan an application modernization.

- **Existing Documentation:** Existant documentation, even if out of date, is an insight into how and why the application was created.
- **Test Plans and Results:** Tests tell you which behavior is important or was frequently problematic.

→ **Run Books:** The requirements to support the application in production are valuable data points.

→ **Defect and Trouble Reports:** These reports tell you how the application was modified after the initial release.

This is far from an exhaustive list, but it will help start the process of documenting what the modernization will need to cover.

Legal and Regulatory Requirements

It's hard to come up with an example of an application that doesn't have to conform to legal or regulatory requirements.

1. **Financial considerations** like Sarbanes-Oxley (SOX)
2. **Privacy laws** such as GDPR, COPPA, CCPA, CPRA, and HIPPA
3. **Local regulations**, including sales taxes

Business Requirements

Beyond regulations are the business-related features required to run the application. These include managing users, billing, inventory management, and accounting. The personnel involved with running the application now have needs and expectations that the new version will need to meet.

User Requirements

Finally, there are the features that sell the product. What's important to your users? What, if it was missing or changed, would make them leave for a competitor? What would they like to see improved?

Considering New Requirements

Are new requirements attached to this modernization? There's an argument for pushing them back until after the modernization is completed, since they could be a distraction.

But there's also a chance that ignoring new requirements will lead to implementing old features that are destined for significant changes or removal.

There's a balancing act involved with considering new features. Which should be implemented right away, and which should wait?

There are a few guidelines you can consider as you review them:

1. **Infrastructure:** Requirements involving infrastructure upgrades or moves, such as new database technologies, shifts to the cloud, etc. should be part of the modernization project.
2. **Legal and Compliance:** Unless the deadlines are far into the future, delaying legal requirements is probably a mistake.
3. **User Features:** User features require careful consideration. Do they eliminate entire pages? Do they require new controls? Adding them may require changing the migration plan.

Potential for Enhancement

An enhancement differs from a new requirement in that it's an opportunity to change the application as a result of the modernization.

Some enhancements are user focused, such as responsive design or proper mobile support. Others benefit engineering, such as a shift from a page-based to a component-based design, or easier deployments via CI/CD or improved packaging. In all cases, these enhancements are part of the project and need to be documented as part of the design.

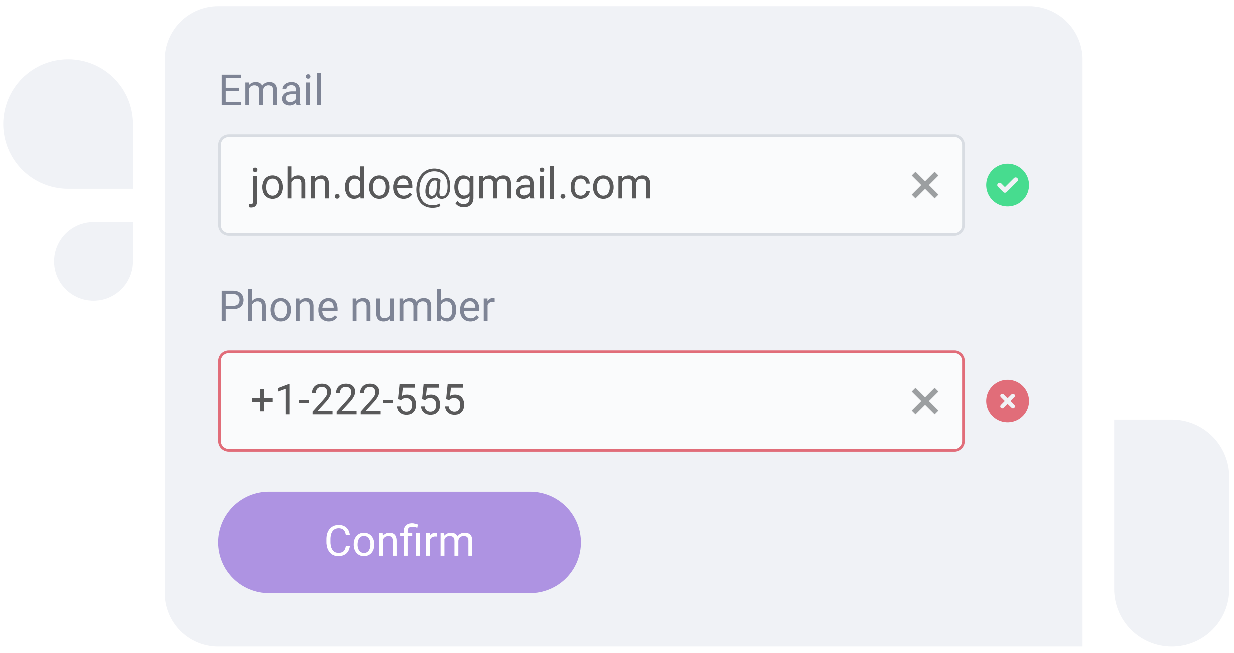
Business Logic Considerations

With new technologies and an altered, if not entirely new, design comes the question of how to transpose your business logic from the legacy system to the new one. Regardless of where you place your business rules in the new design, there will be differences between the new and old tools.

Input Validation

Many legacy applications perform all input validation up to the server in order to avoid users bypassing the front-end business rules by disabling JavaScript. There is still a compelling argument for performing validation on the back end, but the new technologies have differing implementation details to consider.

You can perform syntactic validation in the front end effectively while still leaving critical business logic in the back end, close to your model code. Kendo UI works with JQuery's validation library, so you can use that to perform checks on field contents before forwarding them to the server for semantic checks. Telerik UI has similar support for Blazor's page validation and ASP.NET's Core form validation.



Back End Business Logic

With a modernization that will likely involve breaking a monolith into two or more parts, business logic on the server side will change. Each back-end server or service will need to perform its own integrity checks and implement its own domain logic.

For example, multiple microservices can participate in a single transaction. Each will validate their part of the exchange and can veto the process. **This means your business rules must be atomic and idempotent.** Any service can veto a process, but they won't get another chance if they elect to approve it.

Testing

Modernizing an application provides an opportunity to modernize and improve testing coverage. This is especially important if you plan to take advantage of continuous integration and delivery.

Unit tests also serve as a stake in the ground regarding functionality before, during, and after modernization. If your legacy code lacks adequate coverage, it's often worth addressing that as part of the modernization effort.

Unit Testing Frameworks

Node.js

Node.js has a rich and varied testing ecosystem with over a dozen popular unit testing frameworks. Each supports automated testing; most are useful for front-end and back-end tests.

Unit test frameworks tend to be opinionated, so if you've never used a JavaScript framework before, it's worth looking into a few options before settling on one. According to a 2020 survey by Testim, Jest and Mocha were among the most popular frameworks for JavaScript.

Telerik and Kendo UI

Telerik offers the JustMock testing framework. This unit testing tool supports C# and VB and goes a long way toward providing all you need to develop tests for a legacy front end. Moq is an open-source alternative that offers similar functionality, but it's less comprehensive. Telerik also offers Test Studio, which can test .NET 4.5+, Telerik UI, and Kendo UI applications.

You can test Kendo UI applications with JavaScript tools, including the tools associated with the corresponding JavaScript framework. For example, you can test KendoReact with Enzyme and Kendo UI for Angular with Karma.

Framework-Specific Considerations

In the case of Telerik and Kendo UI-based applications, specific considerations can make or break your development experience, and you should have them in mind.

Telerik UI-Based Applications

When you augment a Telerik component, you can choose one of two approaches.

You can extend it by wrapping it with another higher-order, domain-specific component. This approach leads to effective code reuse but is more involved than the alternative. If you don't take the time and effort to find or design a suitable component architecture and don't set up an effective way to share the code, you won't realize these benefits.

The other option is to patch the component with code-level hooks, such as event handlers, method overrides, and templates. This is faster and produces a component that's tailored to your specific application scenario. But it limits the reusability of the component. You may not be able to plug it in elsewhere in your application.

Kendo UI-Based Applications

For Kendo UI components, it's best to focus on visual customizations via Cascading Style Sheets (CSS). These customizations run efficiently in the browser, allowing Kendo's worker thread to focus on processing messages. This approach also ensures that there's only one type of each Kendo UI component, remaining reusable everywhere. Only CSS classes and IDs need to change based on specific uses, and even they can be reused when possible to save on CSS file sizes.

03

Conclusion

We've covered the business and the technical issues you need to consider when planning to update a Telerik UI-powered or Kendo UI-powered application. These projects cover a lot of ground and involve more than dropping in new libraries and wiring together new screens. They involve new architecture, updated infrastructure, and invariably overlap into business concerns.

But with proper planning and the aid of experienced hands, you can reap the benefits of a modern application that's more reliable and resilient to outages and security threats.

About Resolute

Resolute Software builds responsive, interactive web, desktop, and mobile apps powered by Telerik UI and Dev Tools. We offer comprehensive consulting and software engineering services that are agile, continuous, and delivered on a predictable schedule.

We have years of experience developing and modernizing Telerik and Kendo UI-powered applications.

We've modernized legacy .NET applications that run Telerik technologies. We've moved to new platform applications (e.g., from desktop to web or web to mobile) while preserving their UI capabilities with Telerik components or extending them to add new features when needed.

We can help you choose the technology stack for your next .NET application. We know how to assess your needs and can help you choose between ASP.NET MVC, Razor, Blazor, or JavaScript.

The team at Resolute Software has spent years raising the foundations of the development solutions at Telerik, from UI components for ASP.NET AJAX and WPF to the award-winning Kendo UI suite. As a Progress partner, our connection with Telerik remains solid. We get excellent insight into product roadmaps and the ability to provide customer feedback and influence product development.

— Veli Pehlivanov

Co-founder and CTO
Resolute Software

We also know how and why to construct your .NET server architecture using different paradigms—microservices, monolith, or microlith. Additionally, we design and implement API protocols for client-server communication, taking the data-binding capabilities of the Telerik components into account.

We've helped customers migrate from legacy jQuery-based applications to modern JavaScript frameworks like Angular, React, and Vue while preserving the existing UI functionality and adding new features like responsive design, mobile application look-and-feel, and modern browser capabilities such as caching and data encryption.

We also know how to modernize legacy applications and bring them into the new technological age, transforming them into snappy, responsive web and mobile apps that are always connected, and that work from anywhere, on any device.

Resolute

sales@resolutesoftware.com

Let's talk about your technology requirements.

Get in touch

USA

MA 01701, Framingham,
945 Concord St,

+1-617 386-9697

Deloitte. Company to watch | Tech Fast 50

Business Professional Services

Clutch ★★★★★

People First Company Award 2020 | 2021 | 2022