Characteristics of Modern Web Applications

"... with proper design, the features come cheaply. This approach is arduous, but continues to succeed."
- *Dennis Ritchie*

Modern web applications have higher user expectations and greater demands than ever before. Today's web apps are expected to be available 24/7 from anywhere in the world, and usable from virtually any device or screen size. Web applications must be secure, flexible, and scalable to meet spikes in demand. Increasingly, complex scenarios should be handled by rich user experiences built on the client using JavaScript, and communicating efficiently through web APIs.

ASP.NET Core is optimized for modern web applications and cloud-based hosting scenarios. Its modular design enables applications to depend on only those features they actually use, improving application security and performance while reducing hosting resource requirements.

Reference application: eShopOnWeb

This guidance includes a reference application, *eShopOnWeb*, that demonstrates some of the principles and recommendations. The application is a simple online store, which supports browsing through a catalog of shirts, coffee mugs, and other marketing items. The reference application is deliberately simple in order to make it easy to understand.

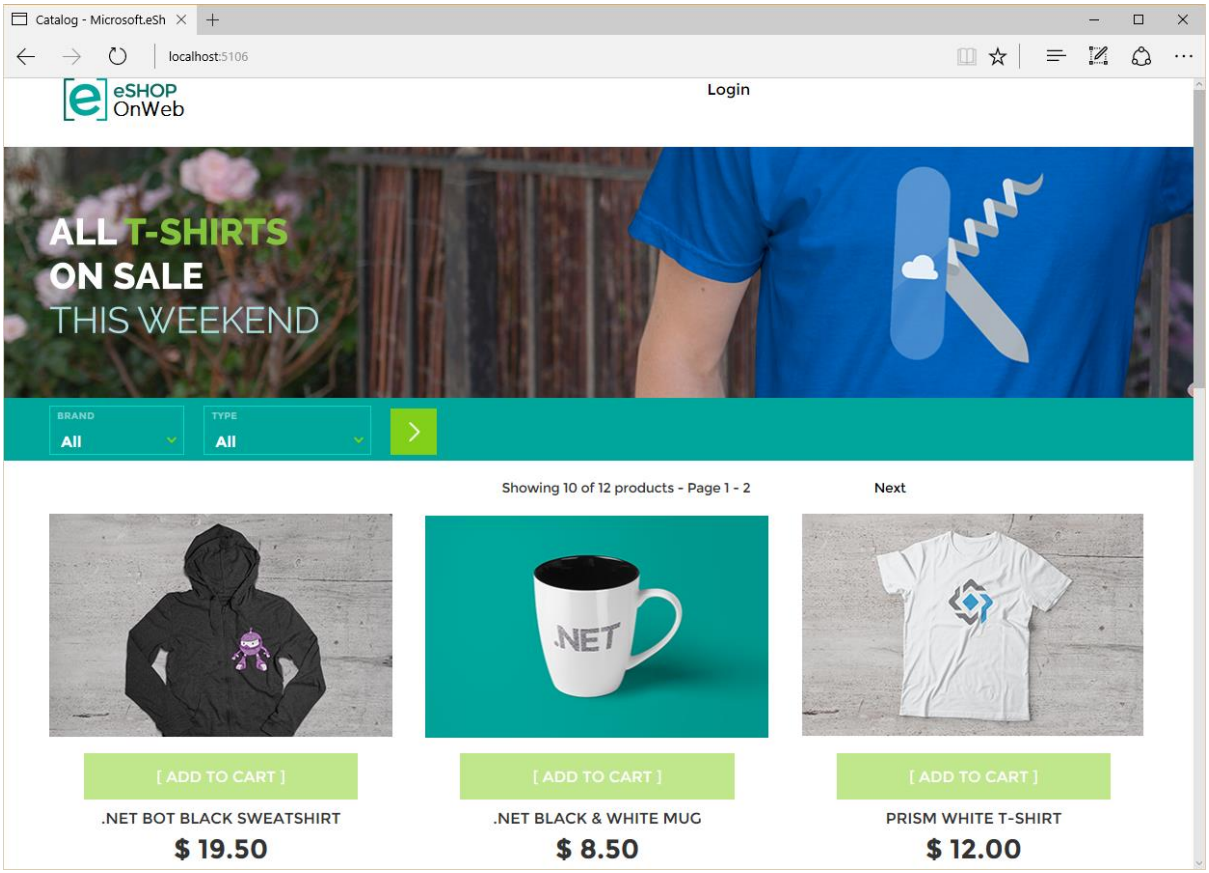

Figure 2-1. eShopOnWeb

Reference Application

- **eShopOnWeb**
  https://github.com/dotnet/eShopOnWeb

Cloud-hosted and scalable

ASP.NET Core is optimized for the cloud (public cloud, private cloud, any cloud) because it is low-memory and high-throughput. The smaller footprint of ASP.NET Core applications means you can host more of them on the same hardware, and you pay for fewer resources when using pay-as-you-go cloud hosting services. The higher-throughput means you can serve more customers from an application given the same hardware, further reducing the need to invest in servers and hosting infrastructure.

Cross platform

ASP.NET Core is cross-platform and can run on Linux, macOS, and Windows. This capability opens up many new options for both the development and deployment of apps built with ASP.NET Core.

Docker containers - both Linux and Windows - can host ASP.NET Core applications, allowing them to take advantage of the benefits of containers and microservices.

Modular and loosely coupled

NuGet packages are first-class citizens in .NET Core, and ASP.NET Core apps are composed of many libraries through NuGet. This granularity of functionality helps ensure apps only depend on and deploy functionality they actually require, reducing their footprint and security vulnerability surface area.

ASP.NET Core also fully supports dependency injection, both internally and at the application level. Interfaces can have multiple implementations that can be swapped out as needed. Dependency injection allows apps to loosely couple to those interfaces, rather than specific implementations, making them easier to extend, maintain, and test.

Easily tested with automated tests

ASP.NET Core applications support unit testing, and their loose coupling and support for dependency injection makes it easy to swap infrastructure concerns with fake implementations for test purposes. ASP.NET Core also ships with a TestServer that can be used to host apps in memory. Functional tests can then make requests to this in-memory server, exercising the full application stack (including middleware, routing, model binding, filters, etc.) and receiving a response, all in a fraction of the time it would take to host the app on a real server and make requests through the network layer. These tests are especially easy to write, and valuable, for APIs, which are increasingly important in modern web applications.

Traditional and SPA behaviors supported

Traditional web applications have involved little client-side behavior, but instead have relied on the server for all navigation, queries, and updates the app might need to make. Each new operation made by the user would be translated into a new web request, with the result being a full page reload in the end user's browser. Classic Model-View-Controller (MVC) frameworks typically follow this approach, with each new request corresponding to a different controller action, which in turn would work with a model and return a view. Some individual operations on a given page might be enhanced with AJAX (Asynchronous JavaScript and XML) functionality, but the overall architecture of the app used many different MVC views and URL endpoints. In addition, ASP.NET Core MVC also supports Razor Pages, a simpler way to organize MVC-style pages.

Single Page Applications (SPAs), by contrast, involve very few dynamically generated server-side page loads (if any). Many SPAs are initialized within a static HTML file that loads the necessary JavaScript libraries to start and run the app. These apps make heavy usage of web APIs for their data needs and can provide much richer user experiences. Blazor WebAssembly provides a means of building SPAs using .NET code, which then runs in the client's browser.

Many web applications involve a combination of traditional web application behavior (typically for content) and SPAs (for interactivity). ASP.NET Core supports both MVC (Views or Page based) and web APIs in the same application, using the same set of tools and underlying framework libraries.

Simple development and deployment

ASP.NET Core applications can be written using simple text editors and command-line interfaces, or full-featured development environments like Visual Studio. Monolithic applications are typically deployed to a single endpoint. Deployments can easily be automated to occur as part of a continuous integration (CI) and continuous delivery (CD) pipeline. In addition to traditional CI/CD tools, Microsoft Azure has integrated support for git repositories and can automatically deploy updates as they are made to a specified git branch or tag. Azure DevOps provides a full-featured CI/CD build and deployment pipeline, and GitHub Actions provide another option for projects hosted there.

Traditional ASP.NET and Web Forms

In addition to ASP.NET Core, traditional ASP.NET 4.x continues to be a robust and reliable platform for building web applications. ASP.NET supports MVC and Web API development models, as well as Web Forms, which is well suited to rich page-based application development and features a rich third-party component ecosystem. Microsoft Azure has great longstanding support for ASP.NET 4.x applications, and many developers are familiar with this platform.

Blazor

Blazor is included with ASP.NET Core 3.0 and later. It provides a new mechanism for building rich interactive web client applications using Razor, C#, and ASP.NET Core. It offers another solution to consider when developing modern web applications. There are two versions of Blazor to consider: server-side and client-side.

Server-side Blazor was released in 2019 with ASP.NET Core 3.0. As its name implies, it runs on the server, rendering changes to the client document back to the browser over the network. Server-side Blazor provides a rich client experience without requiring client-side JavaScript and without requiring separate page loads for each client page interaction. Changes in the loaded page are requested from and processed by the server and then sent back to the client using SignalR.

Client-side Blazor, released in 2020, eliminates the need to render changes on the server. Instead, it leverages WebAssembly to run .NET code within the client. The client can still make API calls to the server if needed to request data, but all client-side behavior runs in the client via WebAssembly, which is already supported by all major browsers and is just a JavaScript library.

References – Modern Web Applications

- **Introduction to ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/

- **Testing in ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/testing/
- **Blazor - Get Started**
  https://blazor.net/docs/get-started.html

Choose Between Traditional Web Apps and Single Page Apps (SPAs)

"Atwood's Law: Any application that can be written in JavaScript, will eventually be written in JavaScript."
- Jeff Atwood

There are two general approaches to building web applications today: traditional web applications that perform most of the application logic on the server, and single-page applications (SPAs) that perform most of the user interface logic in a web browser, communicating with the web server primarily using web APIs. A hybrid approach is also possible, the simplest being host one or more rich SPA-like subapplications within a larger traditional web application.

Use traditional web applications when:

- Your application's client-side requirements are simple or even read-only.
- Your application needs to function in browsers without JavaScript support.
- Your application is public-facing and benefits from search engine discovery and referrals.

Use a SPA when:

- Your application must expose a rich user interface with many features.
- Your team is familiar with JavaScript, TypeScript, or Blazor WebAssembly development.
- Your application must already expose an API for other (internal or public) clients.

Additionally, SPA frameworks require greater architectural and security expertise. They experience greater churn due to frequent updates and new client frameworks than traditional web applications. Configuring automated build and deployment processes and utilizing deployment options like containers may be more difficult with SPA applications than traditional web apps.

Improvements in user experience made possible by the SPA approach must be weighed against these considerations.

Blazor

ASP.NET Core includes a model for building rich, interactive, and composable user interfaces called Blazor. Blazor server-side allows developers to build UI with C# and Razor on the server and for the UI to be interactively connected to the browser in real-time using a persistent SignalR connection. Blazor WebAssembly introduces another option for Blazor apps, allowing them to run in the browser using WebAssembly. Because it's real .NET code running on WebAssembly, you can reuse code and libraries from server-side parts of your application.

Blazor provides a new, third option to consider when evaluating whether to build a purely server-rendered web application or a SPA. You can build rich, SPA-like client-side behaviors using Blazor, without the need for significant JavaScript development. Blazor applications can call APIs to request data or perform server-side operations. They can interoperate with JavaScript where necessary to take advantage of JavaScript libraries and frameworks.

Consider building your web application with Blazor when:

- Your application must expose a rich user interface
- Your team is more comfortable with .NET development than JavaScript or TypeScript development

If you have an existing web forms application you're considering migrating to .NET Core or the latest .NET, you may wish to review the free e-book, Blazor for Web Forms Developers to see whether it makes sense to consider migrating it to Blazor.

For more information about Blazor, see Get started with Blazor.

When to choose traditional web apps

The following section is a more detailed explanation of the previously stated reasons for picking traditional web applications.

Your application has simple, possibly read-only, client-side requirements

Many web applications are primarily consumed in a read-only fashion by the vast majority of their users. Read-only (or read-mostly) applications tend to be much simpler than those applications that maintain and manipulate a great deal of state. For example, a search engine might consist of a single entry point with a textbox and a second page for displaying search results. Anonymous users can easily make requests, and there is little need for client-side logic. Likewise, a blog or content management system's public-facing application usually consists mainly of content with little client-side behavior. Such applications are easily built as traditional server-based web applications, which perform logic on the web server and render HTML to be displayed in the browser. The fact that each unique page of the site has its own URL that can be bookmarked and indexed by search engines (by default, without having to add this functionality as a separate feature of the application) is also a clear benefit in such scenarios.

Your application needs to function in browsers without JavaScript support

Web applications that need to function in browsers with limited or no JavaScript support should be written using traditional web app workflows (or at least be able to fall back to such behavior). SPAs require client-side JavaScript in order to function; if it's not available, SPAs are not a good choice.

Your team is unfamiliar with JavaScript or TypeScript development techniques

If your team is unfamiliar with JavaScript or TypeScript, but is familiar with server-side web application development, then they will probably be able to deliver a traditional web app more quickly than a SPA. Unless learning to program SPAs is a goal, or the user experience afforded by a SPA is required, traditional web apps are a more productive choice for teams who are already familiar with building them.

When to choose SPAs

The following section is a more detailed explanation of when to choose a Single Page Applications style of development for your web app.

Your application must expose a rich user interface with many features

SPAs can support rich client-side functionality that doesn't require reloading the page as users take actions or navigate between areas of the app. SPAs can load more quickly, fetching data in the background, and individual user actions are more responsive since full page reloads are rare. SPAs can support incremental updates, saving partially completed forms or documents without the user having to click a button to submit a form. SPAs can support rich client-side behaviors, such as drag-and-drop, much more readily than traditional applications. SPAs can be designed to run in a disconnected mode, making updates to a client-side model that are eventually synchronized back to the server once a connection is re-established. Choose a SPA-style application if your app's requirements include rich functionality that goes beyond what typical HTML forms offer.

Frequently, SPAs need to implement features that are built into traditional web apps, such as displaying a meaningful URL in the address bar reflecting the current operation (and allowing users to bookmark or deep link to this URL to return to it). SPAs also should allow users to use the browser's back and forward buttons with results that won't surprise them.

Your team is familiar with JavaScript and/or TypeScript development

Writing SPAs requires familiarity with JavaScript and/or TypeScript and client-side programming techniques and libraries. Your team should be competent in writing modern JavaScript using a SPA framework like Angular.

References – SPA Frameworks

- **Angular**: https://angular.io
- **React**: https://reactjs.org/
- **Vue.js**: https://vuejs.org/

Your application must already expose an API for other (internal or public) clients

If you're already supporting a web API for use by other clients, it may require less effort to create a SPA implementation that leverages these APIs rather than reproducing the logic in server-side form. SPAs make extensive use of web APIs to query and update data as users interact with the application.

When to choose Blazor

The following section is a more detailed explanation of when to choose Blazor for your web app.

Your application must expose a rich user interface

Like JavaScript-based SPAs, Blazor applications can support rich client behavior without page reloads. These applications are more responsive to users, fetching only the data (or HTML) required to respond to a given user interaction. Designed properly, server-side Blazor apps can be configured to run as client-side Blazor apps with minimal changes once this feature is supported.

Your team is more comfortable with .NET development than JavaScript or TypeScript development

Many developers are more productive with .NET and Razor than with client-side languages like JavaScript or TypeScript. Since the server-side of the application is already being developed with .NET, using Blazor ensures every .NET developer on the team can understand and potentially build the behavior of the front end of the application.

Decision table

The following decision table summarizes some of the basic factors to consider when choosing between a traditional web application, a SPA, or a Blazor app.

Factor	Traditional Web App	Single Page Application	Blazor App
Required Team Familiarity with JavaScript/TypeScript	**Minimal**	**Required**	**Minimal**
Support Browsers without Scripting	**Supported**	**Not Supported**	**Supported**
Minimal Client-Side Application Behavior	**Well-Suited**	**Overkill**	**Viable**
Rich, Complex User Interface Requirements	**Limited**	**Well-Suited**	**Well-Suited**

Other considerations

Traditional Web Apps tend to be simpler and have better Search Engine Optimization (SEO) criteria than SPA apps. Search engine bots can easily consume content from traditional apps, while support for indexing SPAs may be lacking or limited. If your app benefits from public discovery by search engines, this is often an important consideration.

In addition, unless you've built a management tool for your SPA's content, it may require developers to make changes. Traditional Web Apps are often easier for non-developers to make changes to, since much of the content is simply HTML and may not require a build process to preview or even deploy. If non-developers in your organization are likely to need to maintain the content of the app, a traditional web app may be a better choice.

SPAs shine when the app has complex interactive forms or other user interaction features. For complex apps that require authentication to use, and thus aren't accessible by public search engine spiders, SPAs are a great option in many cases.