O'REILLY

Compliments of Acquia

Creating the Virtuous Cycle with Headless, Hybrid, and Low Code

The Principles of Successful Modern Web Apps

Ronald Northcutt

REPORT

Acquia

EXPERIENCE DIGITAL FREEDOM

Accelerated Application Development

Optimized for both high and low code component-based development, you have the option to deploy **traditional, decoupled, or hybrid** solutions on Acquia's Open DXP.

Discover seamless coordination across IT and marketing teams. **Visit: acquia.com/development.**

Find Out More

Creating the Virtuous Cycle with Headless, Hybrid, and Low Code

The Principles of Successful Modern Web Apps

Ronald Northcutt

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Creating the Virtuous Cycle with Headless, Hybrid, and Low Code
by Ronald Northcutt

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Development Editor: Melissa Potter
Production Editor: Katherine Tozer
Copyeditor: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

April 2022: First Edition

Revision History for the First Edition
2022-04-20: First Release

978-1-098-13055-8

[LSI]

Table of Contents

Introduction

In 1991, Tim Berners-Lee developed the first web browser called World Wide Web and released it to the world. From this humble beginning came a tsunami that reshaped our world in ways that no one could have predicted.

The web browser enabled people from all walks of life to create and consume information from across the globe. This democratization of technology has led to a Cambrian explosion of sorts. Now, anyone can publish content, information, games, community sites, and applications of all types using the browser as a common platform.

One could even say that the web browser is the operating system of the internet. Indeed, Chrome OS has proven that the modern web browser is powerful enough to provide most of the functionality that any user would need. While the internet itself existed long before the browser, it was the browser that enabled everyone to access it.

Today, the browser has become the common system at the heart of the web. Common web standards continue to drive innovation by allowing web applications to experiment with new features and interfaces. We have seen the evolutionary process play out through this dance of competition and collaboration.

Technology has come a long way since 1991. But after decades of building, maintaining, fixing, and architecting web-based solutions, one thing is clear—everything old becomes new again.

While web applications are a specific branch in the broader tree of software development, the basic guidelines and principles behind successful applications have not changed since the foundational beginnings of software development itself. In fact, as complexity has increased, the need for those basic principles has increased as well.

In this report, I attempt to distill over 25 years of experience architecting and building web applications into a series of guiding principles. While I have often acquired this knowledge the hard way (and sometimes at great cost), it is my hope that I can help others learn how to think more holistically about the bigger picture. In other words, take the lessons I've learned and avoid those common pitfalls.

Building modern web applications is not only more critical than ever, but also more confusing. With traditional websites, CMS (content management system) sites, SaaS (software as a service) applications, headless technologies, ecommerce, decoupled apps, mobile, etc., there is no end to the ways that technology is reshaping itself over and over.

The good news is that once you learn to identify the successful patterns described in this report, you can apply them to your projects and prevent problems before they begin. Even better, you can also take advantage of positive feedback cycles to deliver better results faster, with fewer mistakes, and with greater ease.

Web-Based Solutions and Challenges

Building web applications often inspires discussions about hybrid versus headless architectures, or unified versus decoupled applications. Popular trends and frameworks usually drive planning. However, while these decisions are important, there is a much more important understanding that must come first.

We must always remember that we are humans, using human-made tools, to create things for the use and enjoyment of other humans. The trickiest issue at the heart of technology is not the technology itself, but the person behind the tool—how they think, work, and feel about the world.

The fundamental challenge facing architects, developers, and IT leaders today is that the default approach to software development is shaped by basic human understanding. This inevitably leads to the same problems over and over again because the instinctive approach to creating in the digital world mirrors how we create in the physical world. By that I mean that people naturally think about building single, monolithic things. Whether that is a chair, a book, or the next great app, we must first imagine the thing we want to create. However, just as in any other form of engineering, we need to apply systems-based thinking and basic guidelines of good design in order to create things that are scalable, sustainable, and reliable.

Successful web development requires a holistic understanding of the application you are designing (internal systems) and how it fits into the broader picture (external systems). Every system creates feedback cycles that have a net result greater than the sum of those parts. Whether that net result is positive or negative depends on how the system is architected and used in operation.

Your problems will rarely be about the technology used. Your primary problem will always be at the human level. It is not the tools, but the mindset that matters, and that is the secret to repeatable success. When you learn to adopt a component-based approach and the mindset of virtuous cycles, you can be successful with any technology. In some cases, you can even be more successful with old, outdated, or discarded tools because your inherent ability is where the value lies.

In this report, we're focusing on web-based applications that rely on modern browsers as the application framework. However, these general principles for success apply to any form of software development.

In order to break with the pain of the past, we must understand that systems create feedback cycles. These cycles amplify the effects of building and maintaining our applications. Once we understand the difference between virtuous cycles and vicious circles, we can optimize for the results we want.

Vicious Circle Versus Virtuous Cycle

As noted, the default approach to creation tends to focus on a single, monolithic thing apart from the rest of the universe. Often there will be features, requirements, or other acknowledgments that describe the relationship between the thing we are creating and everything else, but these are often secondary considerations.

With a monolithic mindset, everything done to advance or extend the application leads to greater problems like increasing technical debt, bugs, poor performance, and slower delivery times. This is called the *vicious circle*.

When an application is trapped in the vicious circle, most attempts to extend or improve the project actually only exacerbate the underlying problems. Eventually, the application must be decommissioned or replaced before its time.

By contrast, when we pay attention to proper component-based design, we can create workflows that are optimized for more efficient development. Applications can become faster, more performant, and less error-prone over time. They're also easier to maintain, and we naturally develop the ability to pay down technical debt or simply avoid it altogether. We call this the *virtuous cycle*.

The goal is to create applications that can generate the virtuous cycle both within their internal systems as well as within the external systems that they are part of. In order to avoid the vicious circle, we need to understand the primary structures at play and the inherent challenges that come from working with web applications.

Primary Challenges

Most modern web applications are variations on the basic client-server architecture. In the simplest model, the client makes a request from the server, which then returns the requested data to the client. The client then can process that data and make additional requests as needed.

With web applications, the browser is the primary client. However, it often may be making many calls to many servers, and those servers may also be a client making many requests to other servers. These server requests are often organized into "services" or API contracts.

On top of that, the browser is responsible for creating and managing a complex user interface. This means dealing with user interactions, updating itself, and otherwise acting like a proper application should. While the client is often viewed as the "end of the line" in a simplified architecture, a modern web application also acts as an interaction layer itself.

Web applications are based on a collection of different systems that are exposed via interfaces with varying degrees of access and ability.

Successful applications are able to manage this complexity reliably and provide patterns that balance the current needs with future possibilities.

The biggest challenge we face is the ever-increasing number of libraries, frameworks, technologies, and services that are intended to solve these problems. As the number of options grows, it becomes difficult to know where to invest in new tools and where to standardize existing ones.

Choosing new technologies and approaches can give us an advantage, but what happens if those things go away? What happens if they fail, or if they don't scale properly, or if they just don't deliver on the promise?

On the other hand, what about the risk that comes from the lack of innovation? Falling back on old tools and approaches may be faster in the short term, but it does leave us open to disruption by more modern technologies.

This is the fundamental challenge facing IT leaders and architects today. In order to be successful, we have to find a balance between innovation and tradition. Knowing that technology trends rise and fall, we need to be careful not to adopt new approaches too quickly. At the same time, we need to be bold in our willingness to disrupt ourselves and look for better approaches and tools.

The good news is that successful development and architectural principles are universal. These principles work regardless of the specific technologies or the size of the application in question. They can give us the confidence to adopt or reject new technologies because we know that our applications are resilient.

Creating Virtuous Cycles

So, how can an application be resilient? While it is impossible to be "future proof," using best practices will allow you to remain "future ready." The principles that lead to virtuous cycles also put you in a position to make changes in your application without major difficulty.

In fact, when a virtuous cycle exists in an application, the natural tendency is to look for ways to improve, refine, and even disrupt the application from within. The low level of technical debt coupled with a truly composable approach makes experimentation possible, and this is one of the synergistic effects we should expect to see.

The goal of this report is to help you create a virtuous cycle in all your projects. By sharing the guidelines and patterns that make modern web application development successful, we increase the chances of your success and enable you to rapidly evaluate and evolve your tools to meet your organization's needs. The more you practice the patterns of the virtuous cycle, the easier and more natural it becomes.

This is the key secret: there is no secret. It's not about modern technology or the latest advances, nor is it about what some other organization has done to be successful. What works for others may not work for you. This is why we focus on the fundamental basics for all successful projects. When this is done, the appropriate architectural decisions will naturally arise.

Instead of being trapped by the static image and a waterfall-based plan, we look at resource flow and efficiency at all levels of the development life cycle. If you design a system that is efficient and productive for all users, then you will win every time. Good choices lead to positive feedback cycles that reinforce successful patterns. The outcome of every successful project is the virtuous cycle.

Before we can talk about how to set up the virtuous cycle, we need to understand its opposite: what leads to the vicious circle, and what trends, patterns, and habits to avoid if you want to stay out of that trap. Once we are able to identify these things, the principles of successful web application development will make sense, and you will understand how to apply them to your own process.

CHAPTER 1

Vicious Circles: The Well-Worn Path

The vicious circle trap is not new; it goes back to the foundation of software development as a whole. As technology advances with greater power and more tools, it becomes more challenging to build good applications. This is called the *software crisis*:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
>
> —Edsger Dijkstra[1]

The term *software crisis* was coined at the first NATO Software Engineering Conference in 1968 in Germany when the attendees gathered to discuss the new concept of "software engineering" and were surprised to learn that similar issues were plaguing them all. They used "software crisis" to describe the common problem.

The causes of the software crisis were related to the increasing complexity of hardware, and the challenges in adapting the software development process.

1 Edsger Dijkstra, "The Humble Programmer" (ACM Turing lecture, published in *Communications of the ACM* 15, no. 10, October 1972: 859–66).

So, what were the problems they were having?

- Projects running over time and budget.
- Software was becoming more inefficient and error prone.
- Software was frequently failing to meet the requirements for the project.
- Software was increasingly difficult to extend and maintain.
- Some projects failed before they could even deliver workable code at all.

Does this sound familiar? These are the same problems that face many organizations today. The solution to the software crisis was discovered not long after we knew what the problems were, but it is often misunderstood or misapplied. This is actually good news because while computing power has continued to grow, the solution has not changed, proving that it is not actually about the technology.

So, what is this radical solution? It is *component-based design* and *separation of concerns*—the foundational principles of software engineering as we know it. Creating the virtuous cycle and avoiding the software crisis is not a function of time or technology. The basic principles that worked 60 years ago will work today…and 60 years from now.

To overcome the software crisis, we need to understand the ways that poor choices in design and architecture lead to the vicious circle. Software engineers today have more tools and power than ever, and that often creates even more confusion. We can reduce confusion by learning to avoid the worst mistakes and common pitfalls.

Poor Choices

At its core, the vicious circle is quite simple. It's all about poor choices. Poor choices lead to ever-increasing technical debt, more fragility in your applications, and slower development. In other words, poor choices cause pain.

It can be a challenge to identify the difference between a poor choice and a good one, especially if an architect or engineer doesn't have enough experience. Given a long enough timeline over a number of projects, experienced developers will naturally learn most of these principles. Often this is exactly what we see.

Most organizations are faced with a pretty difficult choice. On the one hand, they can spend their valuable resources trying to identify and hire very experienced architects and developers. This is already difficult in the best of times and becomes even more challenging in a competitive market where everyone is looking for the same people. Experienced developers are worth a premium because the investment will save time and money in the long run. They can also accelerate the pace of innovation and unlock new opportunities and advantages.

On the other hand, organizations can try to avoid taking risks and stick to very limited approaches that are more tolerant of less experienced developers. This is often the choice of organizations that rely on a single, broadly adopted technology stack. While this may be a safer course in the short-term, ultimately it will lead to stagnation, and the organization will fall further and further behind. It is difficult for an industry standard to adopt new approaches and avoid disruption. Becoming a fossil is easier than one may think, and given the pace of technological innovation today, it is even easier.

Another option is for organizations to hire architects and developers of varying skill levels. Then they attempt to build applications with varying degrees of ambitious plans, and often with less-than-ideal budgets and timelines. This approach can work if an organization has the ability to experiment and rapidly decommission failed projects. This is a function of organizational tolerance to risk and possible "waste."

Most of the time, however, organizations are unable or unwilling to abandon existing investments. And so, they continue to invest in less-than-ideal projects that accrue technical debt and get more difficult to manage over time.

This process is compounded by turnover in the development team. Employee churn is a fact of life, but as projects become more painful and difficult, they tend to increase the burnout rate of developers. This continues to accelerate the overall degradation of the application as new employees are hired and expected to contribute to a complex and failing project with layers of poor choices embedded by previous developers. Those original developers have long since left, and there is no way to understand why those choices were made. Eventually, the organization is forced to start over and invest in creating new projects, without accounting for those problems or capturing the value of those difficult lessons. The result is that the organization is unable to "learn" from its own mistakes, and so it is often doomed to repeat them.

It is common to approach the new project in a way that reinvents many of the same patterns of the previous application. The business users will often define requirements based on their experience in the previous system. The architects may not have the institutional experience to know the difference between what the users say and what they need.

Repeatedly making poor choices is a major trap and one of the biggest sources of difficulty because it can be tough to break the cycle. Over and over, the organization makes similar decisions, even as they change technologies. The results may be a bit better for a while, but the vicious circle will rise again.

Shiny Object Syndrome

It is tempting and common for architects to look outside for a new tool or approach that will solve their problem. The impulse is to do away with the old and use something new that has advantages that were not available before. This time things will be different, they hope. However, unless they have a good understanding of the principles of good development, then it is just another trap.

Shiny object syndrome, aka *silver bullet-itis*, is when IT chooses the latest and greatest tools and frameworks as the penultimate solution to their problem. This means focusing on new frameworks and tools for their own sake and assuming your problem will be solved by choosing a new tool. This is a particularly easy trap for developers to fall into. It's a natural temptation to want to use new tools and learn new techniques.

Indeed, a good developer will often be on the lookout for new "toys" to experiment with. This impulse often hides the core truth we are discussing here—real solutions are not about the tools we use but the mindset we hold when applying those tools.

This can be tricky because it is easy to look at successful applications and outcomes and mistakenly attribute the success to the framework or tool. This can be exacerbated by sales organizations and evangelists who are tasked with promoting these tools. Unless they have the training to take a "consultative approach," then they will only see the opportunity to sell their tool or ideology. In fact, well-meaning tech enthusiasts are often the guiltiest here. They are so focused on evangelizing their preferred tool that they don't think about what is best for the project itself.

This trap is greatly compounded when developers making the decisions don't have experience using the new technology, which leads to a higher risk of poor choices and flawed implementation patterns. It is tempting to read about the advantages of the powerful features provided by new tools and get swept up in the excitement of what that could mean. This doesn't mean that those tools don't have advantages. What it means is that the problem isn't with the tool itself.

If you can't solve your problem reliably with 10-year-old technologies, then different tools are not likely to help. When properly applied, they may make things faster, easier, or more efficient, but in and of themselves, these new shiny toys cannot make up for bad design patterns.

By way of analogy, if you are a poor tennis player, buying a newer racket is not going to help your game. The only thing that will help you is to focus on training the basics—working on the fundamental aspects of the game, working on your technique, and understanding the strategies and approaches necessary to win. You need to have an open and honest assessment of your own skills and abilities as well as your weaknesses and how you plan to overcome them.

To be clear, better tools might help some things. However, the tool cannot make up for weak skills, inexperience, or a lack of good engineering.

Architecture-First Design

Another related challenge is an architecture-first design pattern. We see this often with trends in the industry. Buzz words like *microservices*, *MACH*, *headless*, and *decoupled* are examples in the web application field.

These architectural approaches can have a ton of value, and often can be implemented in a progressive way. However, without a nuanced understanding of the pros and cons of the architecture, it is difficult to avoid a poor decision. For example, headless systems often provide more flexibility for delivery across multiple channels; however, they also require more development time and more complex deployment setups.

All too often, organizations will choose an architecture based on other successful projects and decide that this is going to solve their current problem. While the success of another application is a useful data point, we must always remember that every project is unique in some way.

This is like using someone else's blueprints to build your house without actually looking at what you need. All architectures are potentially valid, but the appropriate architecture is a result of understanding the needs, constraints, and long-term goals of the project and organization.

You need to ensure that you have the right tool for the job and that you're making good use of the resources you currently have. This may be existing technologies and tools, developer resources, experience, organizational maturity, budget, and other factors. You wouldn't want to use a hammer to put a screw into a piece of wood, no matter how great that hammer might be.

Just like with shiny object syndrome, it can be challenging to understand how to apply new architectures without actually using them. And it is tempting to look at other examples in successful applications and assume that those same patterns will work for you.

A great example of this is Netflix and microservices architecture. Between 2009 and 2012, Netflix refactored its monolithic architecture into a loosely connected series of microservices. This allowed them to continue driving innovation, expanding to over one thousand microservices today. They have also needed to create new tools and services to manage this network, including Conductor (a microservices orchestrator), Simone (a distributed simulation service), and Mantis (a streaming data analysis tool. They have even built their own CDN (content delivery network) and shipped Netflix-specific red servers to internet providers all over the world!

Netflix's success with microservices architecture is one of the most impressive you're likely to see. They have done things and changed things in certain ways that have given them a distinct advantage.

It is worth studying their success in order to understand the value of such an architecture. However, it is a mistake to assume that it is also going to be successful for you. Most development teams simply don't have the size or the scale to apply a similar solution. Trying to replicate what Netflix has achieved with their microservices approach on a smaller scale can often be worse than using a "less interesting" approach.

The trade-off with microservices is that it leads to greater complexity. For Netflix, this complexity is offset by the advantages they get in terms of redundancy, uptime, and self-healing capabilities that enable them to deliver their service with the quality their customers expect.

It is vitally important to remember that your organization is unique. Your team and your resources are also unique. And every project you have, no matter how similar, should be approached with a fresh mind.

Building on the past can be a good thing and disrupting yourself can also be a good thing. But choosing an architecture because it sounds interesting or has been successful for someone else is a short-sighted approach that will lead to more problems than you can realize before it's too late.

Monolithic Myths

Perhaps no concept or word is as widely despised and cursed as *monolith* has come to be. In addition to being a powerful term that characterizes what we may think of when we imagine the vicious circle, it is largely used as a curse or insult. Calling an application "monolithic" is a quick way to make enemies and lose friends.

The monolithic approach is the easiest and most intuitively sensible way to build programs and applications, especially for small teams or single developers. Having all the resources in a single place in the single stack can reduce complexity and increase deliverable time.

Here we find the first "monolithic myth": any degree of unification is a monolithic application. Whether that is the unification of the backend and frontend or the unification of multiple applications in a single repo, many people mistakenly call this a monolith. It is not the unified layer that makes a monolith, it is the *lack of composability*. This means the components that make up the application are not independent and have many cross connections that make them impossible to separate.

Monolithic applications are so common because they are much easier to build. A developer starts on something small and simple, and then slowly adds bits and pieces here and there. For a small project, a single developer (or small team) can hold the entire application in their head, so this can be an acceptable trade-off. For a POC (proof of concept), sample, experiment, or other low priority project, this is not a problem. For production code, however, this is a recipe for disaster. That is why we see such instinctive pushback against monolithic applications. The pattern does not scale.

One problem is that many people equate monolithic with a unified architecture. Having all of the code in one place or having all of the services on one server is not necessarily monolithic. In a true monolith, it is difficult to update or change one piece without disrupting the whole. People may describe it as "spaghetti code" because it is very much like a messy bowl of spaghetti. It is difficult to know where one component ends and another begins, making it challenging to unravel and fix problems. When everything is tied to everything else, it becomes far easier to ignore problems and leave things as they are. Over time, technical debt increases, and the complexity of the application increases at an exponential rate.

To avoid monoliths, you must have a mindset that allows you to build composable solutions. Unified architectures and traditional systems are not necessarily monolithic. By the same token, decoupled, headless, or other "modern" applications are not necessarily composable. In fact, it is incredibly common to see monolithic decoupled applications.

Above all else, we should focus on composability. True composability leads to stable systems that are easier to maintain and upgrade over time. Composability is what makes it possible to limit technical debt and enable us to take advantage of new technologies as they come along.

CHAPTER 2

Composable: Not Just a Buzzword

Composable might be the word of the year, and everyone may claim to have composable solutions, but not everyone understands what it means. As we talk about composability, let's be careful to avoid silver bullet-itis. It is easy to say that something is composable, and easy to expect it may be true. In reality, this may not be the case. So, what does *composable* mean?

Composability is a system design principle that is organized around the relationships of different components. Composable systems provide a collection of components that can be assembled in different ways to achieve your goals. These components are the fundamental building blocks used to build your solutions. As we will see, the principles behind the virtuous cycle rely upon a truly composable architecture.

Composable systems are more resilient because it is easier to evaluate and test each component. This makes these systems trustworthy and future-ready. Adopting new technologies can be done by replacing older components with newer ones. This progressive approach allows applications to evolve and extend their functional life far beyond their original build.

This whole concept of composable architecture and component-based design had its start at the same NATO conference on the software crisis in 1968. The software engineers at the time realized that standardization and modular architecture were as important to creating software as they were to creating hardware.

This concept led to a new higher order of languages and paradigms; and it even continues to evolve today. But the core solution to the software crisis has remained unchanged. People may debate object oriented programming versus functional programming, or MVC (model–view–controller) versus MVP (model–view–presenter), but any of these approaches will work great if they are component based.

Component-Based Design Principles

Component-based design emphasizes the separation of concerns in terms of application functionality. These components may be services, objects, functions, or other logical pieces in a system, but the concept remains the same.

There are several key features to composable components. While we need to make compromises on the perfect component architecture, these guidelines should help. Ideally, the components in a composable system should be:

Modular
: Good components can be added, removed, or swapped out with ease. They will typically follow a standard integration pattern and require little effort to assemble.

Independent
: Components should be as independent as possible. While they may often have dependencies and cooperate with other components, they (and their dependencies) should be replaceable and ideally used on their own.

Stateless
: Components should treat each request or interaction as an independent transaction, regardless of previous interactions. This is why component-based systems will often include a state management system.

Reusable
: If a component is modular, independent, and stateless, then it will also be reusable as well. While this feature may be implied, it is important to call it out. Reusability is a critical feature of composability.

In a truly composable architecture, you can take any component out and replace it without impact on the rest of the system. In practice, this can be a challenge, but if you have a composable architecture, those challenges will be kept to a minimum.

This is a reuse-based approach that allows you to compose software from loosely coupled components. This solves the software crisis by setting up the virtuous cycle and providing benefits for the application as well as the organization itself.

Component Assembly Methods

Component-based design principles take these reusable and swappable pieces and assemble them into different solutions. One of the fundamental decisions that you need to make is what assembly method you'll use for an application.

Historically, assembly methods were what we would consider *high code*. That is, they require a high degree of direct coding or developer involvement to assemble the components into a functional overall system.

More recently however, there's been growing excitement around *low code* assembly tools. Low code tools typically provide some type of interface that makes it easy to assemble components into an application without having to write a lot of code. In fact, with the rise of the citizen developer, we see a whole host of tools that are specifically designed to allow nondevelopers to assemble applications without writing any code at all.

The evolution of software development has been a steady march from high code to low code. The original programs were holes punched in cards. From there, developers moved into using assembly language that was very similar to the machine code it was compiled into. Assembly language gave rise to higher-order languages that provided greater abstractions that were more humanlike to make it easier for developers to do what they needed to do.

And now, you see that this has evolved to the point where, without any development skills at all, people can actually create applications with a high degree of success without knowing any code whatsoever. This evolution is a natural and encouraging trend. It means developers can produce better work in less time.

It is important to remember that both high- and low-code assembly methods are valuable, and necessary. Given the rapid pace of change, we should not be shy about taking advantage of any tool or approach that makes us faster, so long as the results are good. Look at adopting new technologies and architectures where they make sense in the context of the larger project and organizational toolset, and temper this openness with a wariness to adopt new things *because* they are new. This is a delicate balance.

The Prime Directive

The primary guiding principle should be this: we should build composable applications or none at all.

Monolithic applications are the antithesis of composability, and lead to the vicious circle and the pain of the software crisis. While a quick POC or experiment may be created in a monolithic format, it is not a permanent solution.

Truly composable systems are a requirement for the virtuous cycle. They are faster to build and simpler to test. They are easier to adapt and change over time. Most importantly, they lead to the creation of a library or collection of components that can be reused over and over again. This is why composable systems lead to faster development workflows over time.

CHAPTER 3

Virtuous Cycle: The Principles That Lead to Success

The virtuous cycle depends on innumerable factors. Different authors and experts have tried to name them all, and there are many opinions on what the core principles for successful software development should be. There are common patterns that repeat often, and these define the best practices that we see across the board. This chapter provides a high-level list of the key principles to focus on with regard to web applications.

We have already touched on many of these principles, but let's take a moment to explicitly define and discuss them. Many of these may be familiar to you, while others may not be. Still, it is important to focus on these principles in all of your projects. Coming back to the foundational basics is one of the key aspects of success.

Separation of Concerns

One of the most foundational principles in software engineering is the *separation of concerns* (SoC). This is a design principle that divides an application into distinct sections. This results in code that is typically easier to test, understand, and maintain.

Any program that does SoC well will be modular or composable in nature. In order to be truly composable, you need to ensure that the other principles are followed as well. However, simply breaking your codebase into parts or sections will go a long way toward helping you minimize the risk of monolithic entanglement.

There are many approaches to dividing code into sections, such as object-oriented positioning (OOP) and functional programming (FP). There are also architectural patterns like MVC and MVVM (model–view–viewmodel). How you break up a codebase will depend on your needs, the language used, and the preference of the team. The "best" approach is the one that works best for your application and team. As long as you have a good SoC, then the specific strategy is less important.

Think of the User

In all aspects of application development, we should be mindful of the user. This includes not only the end user experience (UX) for the application, but also the future developer experience (DX). There are also times when a "user" may be an external system or outside application that will need to rely on some type of interface (like an API).

Primarily, we want to be sure that we can design programs that will make things easier for the user to do what they need and reduce sources of friction or frustration.

For UX, we want to focus on a clean interface that provides clarity on what to do using common patterns and standards. There are times when you may need to introduce an unknown pattern in order to improve the overall UX at the expense of some frustration, but this needs to be a purposeful choice.

For DX, we want to optimize for productivity for the developer and encourage the principles that lead to the virtuous cycle. In a good project, it is easier to do things the right way than it is to do things the wrong way. This involves project standards, development tools, understanding of the project guidelines, and ensuring there is enough documentation to get a new developer up and running.

Thinking about the user should become second nature for developers and architects. Regular code reviews, user testing, and feedback discussions can help to reinforce and improve the experience for all users.

Keep It Simple, Stupid

In 1960, Kelly Johnson, lead engineer at the Lockheed Skunk Works, handed a team of design engineers a small toolset. He told them that the jet they were designing needed to be repairable by an average mechanic in limited conditions using commonly available tools. Johnson transcribed this directive as *KISS*, and by maintaining this constraint, they were able to create advanced aircraft that were more robust, easier to maintain, and designed to be repaired.

In this context, "stupid" reminds us to avoid overengineering solutions or doing "clever" things that are actually foolish. The most resilient systems tend to be very simple, and so needless complexity should be avoided. This includes not only the code you write, but also the number of services and integrations that are included.

Complexity is a natural byproduct of any project. It will naturally arise of its own accord. So, it is vitally important that we focus on constantly enforcing simplicity. Adding new code and services is easy but removing them can be difficult. Choosing not to include something in the first place is even more difficult.

Simplicity is an important design goal. At every stage, we should be evaluating what should be added and what can be taken away. Refactoring and refining our code should be an ongoing process.

Always remember that simplicity done well is often described as elegant. Truly elegant code is evident by the fact that it is designed to be repaired and extended. This type of code is delightful to use.

Don't Repeat Yourself

Simple code that has good separation of concerns should also be "DRY." That is, we should not waste time and energy redoing the same work over and over again. If a developer is writing code that is functionally similar to something else, then there is waste in the system.

DRY code also encourages reuse and composability. Any functional subsystem should be reusable. This not only saves time in writing code, but also ensures that any changes to that reusable code will be leveraged everywhere it is used. Building a collection of modular components allows you to reuse your existing work both when creating new code as well as when improving or refactoring existing code.

A general rule of thumb is that if you are building something more than once, then it needs to be standardized and reusable. Having a library of utilities, helpers, API interfaces, and other tools is the typical result of DRY programming.

Abstraction

As your application grows in complexity, it will naturally develop certain abstractions and layers of separation. Often this begins as utility or helper functions but can become entire patterns or subsystems in larger applications.

The goal of abstraction is to suppress some of the complexity from the user and maintain a simpler interface that gives the user what they need. This abstraction can take the form of interfaces or conventions that enable standardization of common patterns. This can not only make it easier for the user but also open the door to more advanced functionality in the future.

A great example of this is a database abstraction layer. On the surface, this would simplify the way a user interacts with the database, making it easier for them to quickly access the database in the most common ways.

A database abstraction layer also creates other opportunities for improvements. This abstraction layer can include protections against common database attack vectors or sanitize data before it is inserted in the database. It can also provide a more restricted security access layer to the database itself. It could be extended to provide similar access to other data sources.

With web-based applications, API services often act as an abstraction layer within the architecture. Even here we may find value in using a library or utility to manage the interaction with that service. This is why it can be tempting to focus on adding microservices and APIs to our application stack.

One warning—while there are great advantages to abstraction, we must be careful to avoid the temptation to add unnecessary complexity. Always remember to focus on simplicity first. When an abstraction layer provides a simpler method for working with and maintaining systems, then it is usually a good fit.

Single Responsibility Principle

If components are simple and DRY, then they will often be restricted to a single responsibility. Every component in a modular system should focus on its own part of the whole, often to a narrow degree. Your functions, methods, and components should really do one thing, and do it well.

Keep in mind that in a composable system, you will often have “super components” that are composed of other components. Your more atomic components should be simple and clear, but your complex super components should also be clear in their responsibility.

This encapsulation of functionality is a further extension of the SoC principle. Whenever you refactor or review code, it is good to look with the single responsibility lens. Is this component doing “too much”? Can we simplify its operation and minimize complexity by breaking it up?

Some developers like to adopt rules about maximum lines of code that a component (object method, function, etc.) can have. While not necessary, it can be helpful to have guidelines for a team or project. The single responsibility principle leads to simpler components that are easier to read, understand, and test.

Comment Your Code

One of the problems that leads to poor choices is lack of historical context as newer developers wrestle with code that someone else wrote. To be fair, this also affects more experienced members of the team, and even the developer who wrote the code may have forgotten the details about it!

In any case, the solution is the same—document the code. Documentation describes how the code is expected to work, how to use it, and often why certain choices were made. Good documentation also includes examples, where necessary, and is useful for helping future developers in learning how to use or fix what was done.

Sometimes it makes sense to maintain documentation outside of the code, especially for utility libraries that may be used across many projects. However, the default instinct should be to provide documentation *inside* your code. Typically, this is in the form of inline and more formal docblock comments.

Every method, function, class, or other component should have comments in code. *Every single one.* This not only helps to preserve the knowledge about those components but can be invaluable later when there is a need to fix or refactor the code. Good comments will also include notes to future developers about places for improvements or other "to-do" items that could be valuable later.

Make documentation part of your coding requirements and look for common standards for commenting. Most of the time, you can also get the best of all worlds by adopting commenting formats that can be parsed to generate documentation outside of the code later. This habit is one that all developers should adopt, and all organizations should expect.

Writing good comments and documenting your components is an investment in future efficiencies. The payoff will come in the form of fewer bugs, more clarity in your code, and greater reuse overall.

Generality

Optimizing components for reuse and following the DRY principle will often lead to a degree of generality, ensuring that your software is flexible enough to work in a variety of situations. Typically, this means removing artificial restrictions or overly specific assumptions to encourage reuse.

A great example of this is the Y2K problem. In the late 1990s, there was a growing realization that a significant body of software had been created to store year data in two-digit format. This meant that when the year 2000 came around, much of that software would register the "00" and "think" it was 100 years earlier.

The default assumption the developers made was that the software was running in the 20th century. There was quite a bit of concern about the potential problems with this limitation, from banking to power grid failure. Many organizations were forced to invest in large-scale projects to refactor or "fix" their software, sometimes with major difficulties.

For those developers who stored time in a less restrictive format, this was no concern and their more general approach was seen as a wise and sustainable choice.

In hindsight, it may be difficult to understand how anyone could fall into such an "obvious" trap. However, this is still a problem to this day. Many programs store time and date data using *Unix time*, which is a 32-bit register measuring seconds since 1970. The problem with this approach is that this will fill the 32-bit space in 2038, leading to a repeat of the Y2K problem.

While these are specific examples, there are many times when application decisions and assumptions today can cause problems tomorrow. Good developers know that it is useful to provide enough generality in the system to accommodate for unforeseen issues that may pop up later.

Perfect Is the Enemy of Good

Sometimes the best choice is the worst choice. Good software design is about making compromises based on time, budget, requirements, and developer ability. Most projects begin with lofty goals that are diminished or watered down over time.

This process is actually a good thing. If we were to build the most expansive and impressive software we could imagine, then the work would never be complete. Most of the time, the ideal situation is to make good choices to achieve a good result rather than waste resources attempting to achieve perfection.

Perfection-seeking can take many forms. Sometimes it is over-optimizing each phase of a project instead of completing things with reasonable inefficiencies. Sometimes it is an obsession with "pixel-perfect" design results. Sometimes it is simply discussing or debating the plan for weeks and months instead of actually getting to work.

Remember that the last mile to perfection is the longest. Focus on creating the virtuous cycle and allow good results to become great results. The strive for improvement is noble, but the focus on perfection is self-defeating. Simplicity, speed, and resilience are the hallmarks of good software. Perfection is an illusion.

Consistency

Creating software is often a team effort that takes place over months or years. With so many people contributing to the codebase over time, it is easy for different developers to introduce their own preferences on how the code is written. This can lead to major inefficiencies because each piece of the application might follow different paradigms or have different expectations. A lack of consistency across the applications can be a massive drain on developer performance and a source of unexpected bugs and friction.

The easiest way to avoid these problems is to adopt consistent standards across the project, including basic things like code formatting and commenting conventions. This can also include more advanced opinions about things like recursion, loops, or other software patterns. Automated code review tools can be included in the workflow to find and optionally fix areas where inconsistencies have crept in.

Over time, most applications will naturally increase in size and complexity. Enforcement of project standards helps to maintain consistency and will keep the code cleaner and more efficient.

Open Source

One of the fastest ways to build applications is by utilizing open source technology. Open source means that the code is available to be viewed or changed, and often has a free and permissive usage license.

It is common for developers to accelerate projects by assembling existing open source code into a new project. In addition to getting "free" code and support, common open source projects often have additional resources in the way of documentation, code samples, and even entire projects.

There are considerations to keep in mind, however. There are thousands of open source projects, frameworks, and communities to choose from. Some of these may not have good security or performance considerations or support. Choosing the wrong open source project can sometimes increase your technical burden.

Understanding how to evaluate and find good open source projects could be the subject of an entire report; following are some tips to consider:

- Look for projects that have history. Active projects that have been around for a while usually have a track record of success.
- Look for projects that have a broad user base. More active users means that the code will likely have more updates, more eyes, and a longer future.
- Pay attention to the community culture. Communities with an open and inclusive culture are more likely to grow and stay healthy. Restrictive or negative communities will drive away good developers.

It is impossible to build web applications today without reliance on some open source code. Even completely custom code is dependent on browsers that are largely open source in nature. It is important to understand how to leverage open source code for your projects.

As a final note, keep in mind that part of using open source code means contributing. Sometimes that is as simple as issuing a bug report or testing a patch. However, it is always useful to look for ways to contribute time and code back to the community.

Remember, it is easier to contribute 5% of what's missing than build 100% of what you need. If you follow the principles of modularity, generality, and consistency, then you can find many opportunities to give back to the community. The upside is that you also get more people using, testing, and contributing to your code as well!

We can get more results over time from less work—the rising tide lifts all ships. This is a very clear example of the virtuous cycle at work.

Incremental Development

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

—John Gall[1]

One of the major problems with the vicious circle is that it becomes nearly impossible to make changes over time. Incremental development helps avoid this problem when combined with component-based design principles.

When you take an incremental approach, there is a consistent addition of small bits of functionality. Once you have tested and accepted each new component, you can safely move on to the next one. Over time, complexity will naturally increase as the system becomes more robust. However, this growth is sustainable and manageable.

Agile software development is very much in alignment with this principle. However, even waterfall methods of planning can be organized to follow an incremental process. In any case, priority should be given to the simplest use cases that provide for high value features for a minimum viable product (MVP). More complex components that have other dependencies or are more likely to change can be moved to later in the development cycle.

Everything great first begins as something small. Starting simply and focusing on the fundamentals ensures that the project can continue to grow and evolve over time. This approach also makes it easier to maintain a balance between the resources you have and the goals of the project.

1 John Gall, *General Systemantics: An Essay on How Systems Work, and Especially How They Fail* (General Systemantics Press, 1975), 65.

CHAPTER 4

Resource Flow and Efficiency: Balancing Business and User Needs

Following the guidelines outlined here will go a long way to helping you create the virtuous cycle and successful web applications. However, there will always be a need to find compromise between the goals and the budget.

Often, an organization will look at only the initial costs or maybe the development time as the primary resource. This can be helpful but leads to challenges because we must think about the users. The up-front cost is not the only resource to consider.

Focusing on the proper and efficient use of resources is incredibly important. If this is done at each stage, and the principles are applied, then everything else will take care of itself. In fact, many considerations like tech stack or architecture won't really matter if you have an efficient and workable workflow. This ensures that resources are used wisely, and in a composable application, changes or adaptations are always possible.

As noted, we also need to consider the end UX as well as the DX. Balancing the needs of the whole workstream is important in order to not only create but also maintain the virtuous cycle. When we focus on the UX and DX, we see an *inverse* relationship between the user's resources and the developer's. Many organizations will make the mistake of paying attention to only one or the other, but both are vital.

As we make the UX better, the DX will suffer, and vice versa. This is where compromise comes into play and the ability to take a holistic view of the entire project life cycle. When we find the right balance of bandwidth, time to market, and developer time, we can maximize the virtuous cycle and ensure the highest return on investment for everyone.

Bandwidth

Bandwidth is high value for users and low value for the business.

The cloud has made bandwidth and processing power relatively cheap compared to the "old" ways of on-premises servers and IT management. Now, anyone can spin up a cloud server, cloud storage, or any number of software as a service or platform as a service providers to build and serve their application. Broad access to CDN technology reduces the cost even further, potentially reducing the business cost for delivery to incredibly low amounts.

However, this low cost on the delivery side makes it easy for organizations to build inefficient code that uses more resources. This inefficiency is paid for by UX—time to first paint, device memory, load times, connection speed, latency, etc., which can have a dramatic impact on the end experience and kill an otherwise great application.

Customers expect fast responses and simple interfaces. They expect access on mobile devices with smaller screens and slower connections. A web application that loads in 1–2 seconds on a computer can take 10 times as long on a mobile or IoT (Internet of Things) device, frustrating end users and driving them away.

We must always think about the user and ensure we are building great experiences. As much as possible, we want to use server-side rendering (SSR) for pages and content, and cache it at the edge. This can be further augmented by a static site generator (SSG) where it makes sense, though we want to be careful not to overoptimize for UX at the expense of DX.

Above all, keep the size of payload as low as possible in order to conserve the bandwidth of the user. Some strategies include:

- Aggregate and minify JS and CSS aggregation and minification.
- Create JS collections to package and remove unused code from pages.
- Compress your assets so they are as small as possible.
- Use lazy loading techniques to only load larger assets when they are needed.
- Follow progressive web application strategies for preloading common assets.
- Rely on local caching to avoid making duplicate requests.
- Enhance your application with offline access to ensure your application is always available once users have loaded it.
- Limit the amount of JS parsing on the client.
- Consider using preprocessing techniques to keep client-side code fast.
- Avoid using heavy JS frameworks unless absolutely necessary.
- Push as much JS to the bottom of the page to keep it from blocking initial paint.
- Remove or delay third-party JS libraries that will slow down your page.
- Use responsive web design to avoid loading large assets on smaller devices.

Designers want big images and videos. Marketers want tracking and analytical data from across their toolset. Developers want cool features and app-like interactions. All of these are easy to put into a web application, and all of them carry a fairly big cost. Giving everyone on the business side what they want will cause the user to pay the price.

Be ruthless in your efforts to keep bandwidth "costs" low, and be creative in ways to limit, delay, or hide the loading of assets so they don't block the user's experience. Your application will be faster, your UX will be enjoyable, and your customers will thank you.

Time to Market

Time to market (TTM) is a middle value for both users and the business.

Value is only realized when it is delivered. Great information, features, or integrations in your web application can only affect the user and the business only after they are deployed and available.

So TTM, or value throughput, is an important consideration. A large benefit of the virtuous cycle is that it enables the business to provide faster and more consistent delivery of value. This is accomplished by removing bottlenecks and other impediments to workflow.

For the business, this enables them to get out new messaging, release new features, and deliver other assets to the field as quickly as possible. With a good workflow, the creation of these resources can immediately flow into delivery with little to get in the way.

For the end user, faster TTM means they will get those features and resources quickly. Users expect instant access to information and continuous feature releases and improvements.

To increase TTM, look for opportunities to leverage low-code tools to empower business users and provide self-service opportunities to create and deliver work as efficiently as possible. Ensure that you are decoupling content and experience management from code deployment where possible.

Where the high-code approach is necessary, be sure to invest in continuous delivery processes to automate building, testing, and deployment of code. Composable workflows will also help to reduce rework and leverage existing code.

These concepts are about parallelizing the workflow to remove serial bottlenecks and move delivery as close to creation as possible. The goal is to get a quick turnaround from creation to delivery so you can get feedback quickly as well. This is part of the virtuous cycle because it makes it easier to rapidly evolve the application and adapt to changing requirements.

Developer Time

Developer time is low value for users and high value for the business.

Developer time is one of the most valuable resources that a business has. These people are vital to ensure the proper building, maintenance, and uptime of the web application. The more experienced they become, the more valuable they become as well.

Unfortunately, the end user doesn't need to care about the cost of developer time. As a consumer of the application, the cost to build the application is largely abstracted away. The only real way that developer time impacts the end user is when it impacts features or TTM.

Increasing DX means investing in the tools and processes to make it easier for developers to be productive. Continuous delivery practices can enable automated testing, linting, building, code-quality scans, and other processes that can enhance developer productivity. These can also be used to enforce best practices and ensure consistency across the team.

Look for ways to reduce the distractions for developers. The typical developer only spends about 35% of their time writing the code. Automation can help free up time, as can following agile practices and reducing extraneous meetings.

Also look for ways to let developers focus on high-value activities. Low-code tools can enable nondevelopers to control some assembly options and keep developers from using their time to arrange components on a page. Developers should be building, testing, and deploying components. They should be researching new technologies, experimenting with innovative ideas, and improving deployment processes.

Standardizing on a common toolset and managing that toolset will be vital to optimizing developer time. The goal should be to have systems in place to enable a new developer on a project to be up and running with an optimized development environment, process, and workflow in 30 minutes or less.

If you can't standardize the development experience and workflow across the team, then you can't enable good flow. At its core, the virtuous cycle is about inducing smooth and efficient workflows. Good DX is vital to the success of any development team.

CHAPTER 5

Bringing It All Together

Many organizations struggle with building successful web applications because they get trapped in the vicious circle of development known as the software crisis. In an attempt to escape the trap, teams may try to look for a silver bullet or some technology that will finally save them.

Technology is not the solution to the problem. The real secret to the vicious circle is the mindset we must adopt to instead create the virtuous cycle. When this cycle is created, technical debt is reduced, throughput is increased, and the workflow becomes more efficient over time.

The core architecture for creating the virtuous cycle is composability. Successful teams know that we must avoid the monolith by building components and choosing the best assembly method to package and reuse those components in reliable ways.

There are also distinct principles that software engineers should keep in mind in order to keep their projects on track. These principles are useful to any application development process and can be applied at multiple stages.

Your user cares about the experience, which is largely impacted by bandwidth and delivery options. The business cares about developer time as a valuable resource and how to set up the right workflows to maximize efficiency and workflow. These priorities also help both the user and the business maintain throughput and a fast time to market.

Innovation is important and evaluating new technologies can provide new opportunities to disrupt the status quo. However, everything old is new again. A thorough understanding of the principles of good web applications will allow even old technologies to win against the latest and greatest.

Textbook Example

So, what does an amazing web application look like? We have a great example making waves right now—Wordle.

Wordle is a web application created by Josh Wardle. It's an elegant game that provides a daily word that must be guessed in 6 tries or less. The game features a simple interface that tells the user which letters are correct or wrong based on colored boxes.

Josh created this web application as a gift for his partner, who loves word games. It has become a bit of a viral phenomenon, had some improvements over time, and was sold to The New York Times for a "low seven figures."

While the creativity and simplicity of the game itself is part of its appeal, as a web application, it embodies many of the principles and recommendations in this report:

KISS
: The game is very simple overall, and while it does have some settings and additional features, they are also limited. There was no plan to monetize or advertise in the game, and it was always intended to stay simple.

Standardization
: The game is built using vanilla JavaScript and native web components.

Low tech
: There are no fancy frameworks or libraries used to build the application. Much of the elegance of the application is due to its reliance on commonly accessible features in the browser, like local storage and custom components.

DRY
: As a component-based application, it is very DRY.

Incremental development
: Josh has talked about how he made changes to the application over time, even adopting features (like the share) from user feedback.

Stateless
: The game ships with all the code and data it needs and maintains state in the browser. This means that every user has their own experience and stats, but there is no need for the code to manage that state directly.

Additionally, this is a very efficient application. It is relatively small in size, so it can quickly and easily be downloaded by anyone in the world. The code is cached locally to some degree, so subsequent reloads each day are superfast. This makes it infinitely scalable because there is no heavy backend to maintain, and the application is self-contained. Millions and millions of people all over the world play this game every day, and yet the delivery costs are almost nothing.

The application was built by a single developer who didn't need a massive team or an incredible amount of time. It was also based on some earlier experiments Josh tried out, showing that some degree of standardization and reuse are at play.

As an application, Wordle is much simpler than what most organizations will want and need to build. However, it is a beautiful example of the principles of the virtuous cycle in the real world.

This application can live for years and years without any additional investment and could also be the basis for other games. Each variation would have been easy and fast to create, additional features could be added with minimal difficulty, and all of this potential exists because it is a composable application that embodies the virtuous cycle.

Where to Go from Here

This report is purposefully vague about *specific* technologies and architectures that can be applied. The goal is not to tell you what to think and do, but *how* to think. It should be a useful resource for discussions about your current and upcoming projects. It can also help provide a common language for describing problems, solutions, and objectives.

Some parting advice:

- Be wary of new technologies but be excited about new possibilities. There is no silver bullet out there, and new technologies should be evaluated on their own merit. There is no need to worry about missing out.
- Focus on the common standards and traditional approaches and learn how to properly apply them following the principles in this report.
- Mind your resource workflow and focus on UX as well as DX. Look for ways to be "lazy" and reuse code, whether that is from internal component libraries or external open source repositories.
- Keep your wits about you—the software crisis is ever present. A composable approach to application development coupled with a focus on elegant simplicity will save you and your team.

The secret is to invest in developing your mindset. Everything else will flow naturally from there.

About the Author

Ron Northcutt is the director of developer advocacy at Acquia, where he started as a solutions architect in 2014. In that time, he has helped contribute to tens of millions of dollars in revenue and numerous success stories inside and outside of the company. Interacting with so many large-scale projects has given him unique insight into application design.

Ron is a professional problem solver with a long history of diverse experiences. From ranch management to real estate brokerage to solutions architecture, his 25 years of experience in web technology has allowed him to see multiple cycles in the industry, as well as the underlying patterns that form successful projects—and the ones that form dismal failures.

He is an avid supporter of open source software, and a frequent contributor to several projects, especially Drupal. When not at work or online, Ron attempts to play guitar and rock climb. His children do a good job at keeping him humble on all accounts.