
Universal
Design
Systems

Overview

We are surrounded by seemingly invisible forces that guide the creation of everything from home appliances to digital media. These forces are often standardized into a set of rules that ensure consistency and reproducibility in the creation process. One such set of standardization can be found in the digital products we use everyday. These are considered style guides or design systems, and they underpin every visual and functional decision of the product.

A design systems prime directive is to provide a stable and reliable source of product information. However, with today's digital product design ecosystem, these systems are starting to struggle under the weight of shifting product requirements and an ever changing digital landscape, falling short of their original goals. Trying to maintain stability in such systems has become a costly and frustratingly inefficient experience. When agility and adaptability are what sets apart the good systems from great we must find a way to imbue our systems with these capabilities.

To achieve this we can look at the concept of the Universal Design System (UDS) – an approach to design systems that breaks free from the constraints of traditional system frameworks. “Universal Design Systems” delves deep into the significance of two primary components: Agnostic Design Systems (ADS) and Product Kits (PK), a dynamic duo that operates better together than alone. The combination of these two will afford us with a flexible, efficient, and understandable design system.

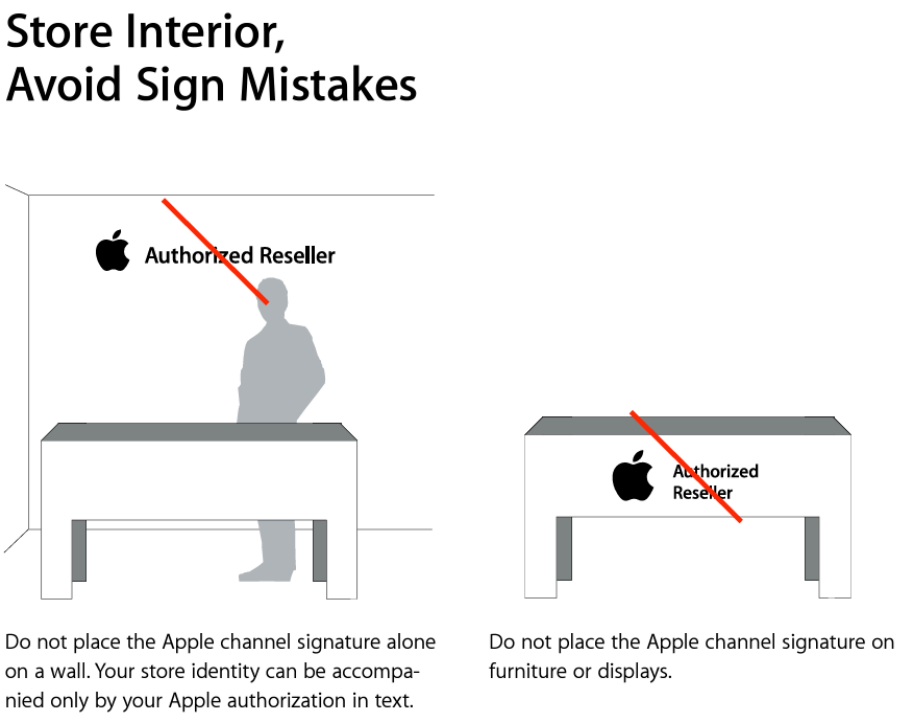
By the end of this journey, you'll gain an understanding of the importance of the Universal Design System and the guiding principles behind it, illuminating its positive impacts on both the end user and product maintainers. Perhaps you'll even be inspired to apply what you've learned to an existing product or project.

Table of Contents

Chapter 1: Design Systems

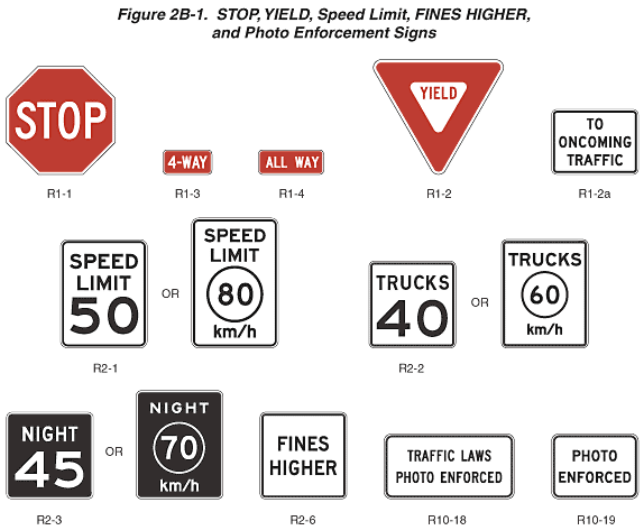
Design systems govern almost all of the products we use. They are a silent mediating force that helps ensure that visual and functional details are in alignment. Let’s examine what a design system aims to accomplish.

Design systems are just what they sound like: a systematic organization of information that enables brand adherence and feature reproducibility. They contain the intended colors, icons, functionality, and more that make up the fingerprint of our everyday products. These systems don't start and end with digital experiences; they also appear in the physical world as style guides. These are the blueprints that guide the design of shampoo bottles, cereal boxes, home appliances, etc. With the system acting as guidance, brands become easier to recognize and their product experience becomes more predictable. Here we can see that within Apple's guidelines for store identities Apple provides explicit instruction on how store owners should and should not display signage. This is just one of many guidelines that Apple provides and it helps to ensure a cohesive and consistent experience at all Apple store venues.



Apple Identity Guidelines for Channel Affiliates and Apple-Certified Individuals (page 50)

Even everyday utilities rely on a standardized ruleset to remain effective. In the United States, stop signs appear almost the same wherever you are. According to the US Department of Transportation's "Manual on Uniform Traffic Control Devices (MUTCD)", Chapter 2B "Regulatory Signs", Section 2B.04 STOP Sign (R1-1), stop signs must follow these standards.

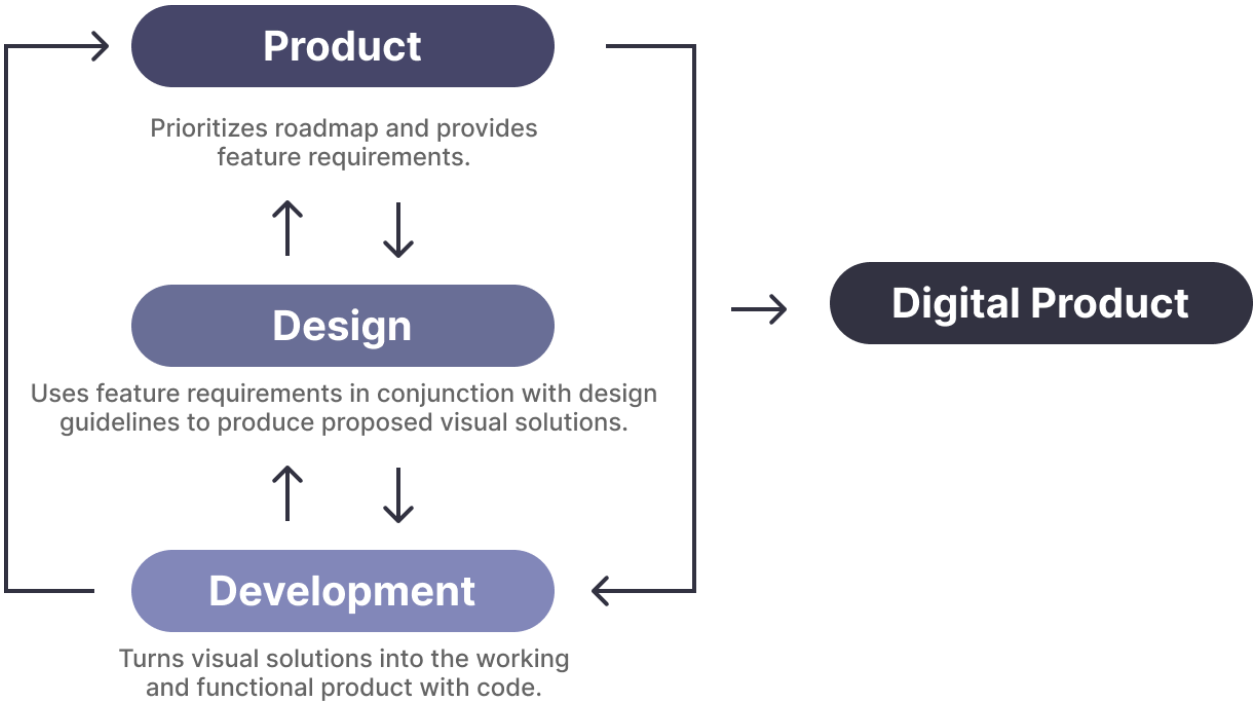


DOT MUTCD - Figure 2B-1

1. The STOP sign shall be an octagon with a white legend and border on a red background.
2. Secondary legends shall not be used on STOP sign faces.
3. If appropriate, a supplemental plaque (R1-3 or R1-4) shall be used to display a secondary legend.
    a. Such plaques (see Figure 2B-1) shall have a white legend and border on a red background.
4. If the number of approach legs controlled by STOP signs at an intersection is three or more, the numeral on the supplemental plaque, if used, shall correspond to the actual number of legs controlled by STOP signs.

This set of guidelines is rigid enough to allow stop signs to communicate a predictable and reproducible message to all drivers while allowing flexibility in niche situations requiring modification. Drivers, much like users of digital products, have learned to expect specific outcomes from their previous experiences. These standards help not only the driver but also the individuals creating new road infrastructure by ensuring consistency. Imagine if a road worker was feeling fun and turned a bunch of stop signs into white triangles; drivers would probably pass them off as an unimportant oddity and keep driving. Similarly, users of digital experiences have come to expect various mirrored experiences between the products they use. Swiping your finger should scroll the page, trash can icons usually imply some sort of delete feature, and buttons perform some action. Much like the Department of Transportation, digital products require their own standardized documentation that helps instill a sense of trust within the user experience.

Physical style guides and digital design systems tend to have many parallels, but the digital systems' environment requires the incorporation of supplementary information and processes. The majority of this information comes from three main groups with a hand in the product.



The development, design, and product stakeholders provide the bulk of resources that make up the final product. These are the three internal groups that a design system needs to cater to along with the external end users of the product. Similar to how the DOT's standards are intended for both the drivers experiencing these road markings and the ones implementing said markings. Design systems serve stakeholders, who must work from a set of standards to effectively produce the features and experiences required by the product, and the user, who relies on a consistent and predictable experience.

In essence, the product stakeholders focus on the high-level needs and desires of the product's target audience. They assess roadmaps and form the feature requirements that describe what should be incorporated into the product. The design and development stakeholders will utilize these requirements to piece together the solutions that end up getting implemented into the final product. Designers then generate visuals and user experience flows off these requirements. In the end designers need to provide well thought out and detailed guidance for developers to

work from. Developers will take these design specifications and incorporate them into the code of the product.

Throughout this process each group is having back and forths with one another to ensure accuracy to the experience of the existing product and adherence to the requested features. In order for teams to reliably produce new features there has to be a shared set of documentation that keeps everyone on the same page. To keep the product visually and experientially consistent, designers should have a set standards for styles, components, and experience patterns. Without this you run the risk of having 5 different ways users perform the same action or different ways of visually portraying the same content. By establishing set ways of serving up experiences we ensure consistent outputs. Developers require similar documentation in order to keep the creation of repeat code which causes maintenance issues and increases time spent in the development phase. Documentation such as templated component code and code patterns keep cross-team work in sync and repeat work down. Product stakeholders are a great example of the need for shared information systems. They are in the unique position where a solid understanding of the availability of components, experience patterns, and styles enable them to more accurately assess the possibility of their feature requests and timeline expectations. Product individuals can compare product needs against what exists today. They can see that certain components are available to solve for their needs. On the flip side, they will see that the components needed for their features do not exist, affecting timelines and resource allocation.

Together, each group should work together to keep each informed through the creation of standardized patterns and documentation that is shared throughout the organization. Without this, the disjointed nature of the product would be very apparent as each individual contributor starts fresh with each new assignment.

These documentation sets are often where many issues within design systems lie and it’s not just the documentation itself that is the problem. The combination of neglected documentation and a system whose foundation lacks a framework conducive to properly maintained documentation is where the true issues are found. The complexity of digital products and the landscape they inhabit is requiring design system solutions that can match that complexity and scale.

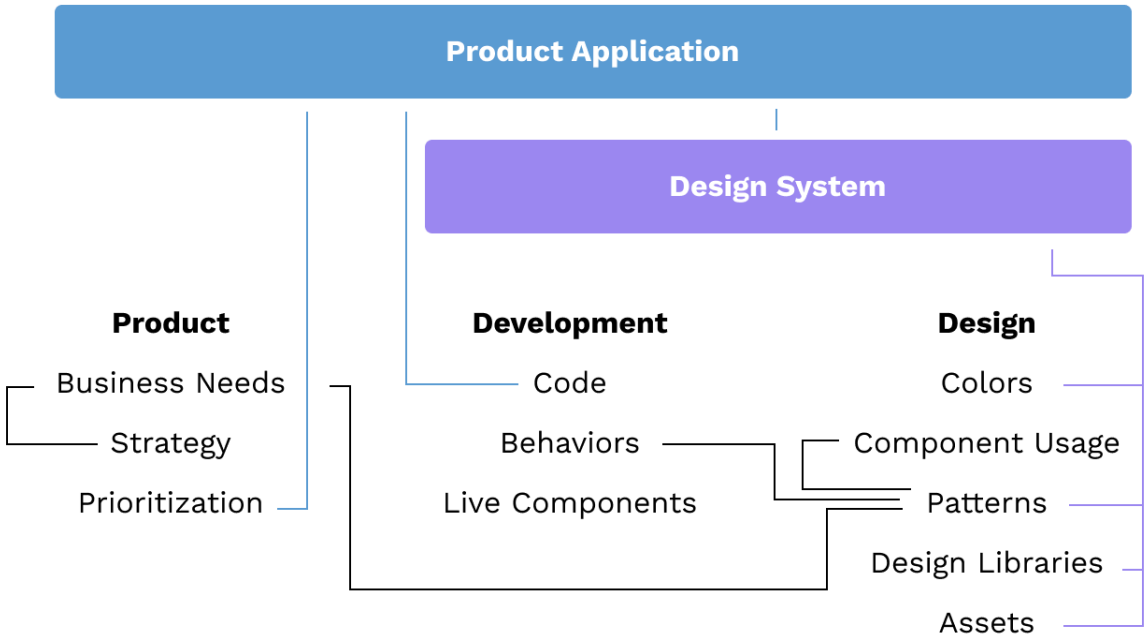
Chapter 2: The Landscape of Systems Today

Now that we know what a design system entails, we can dive deeper into the shared methodologies and frameworks that today's systems operate on. We will examine how such systems have led us to a crossroads of inefficiency and frustration. This chapter covers the current state of affairs, exploring the dilemmas that plague traditional design system architectures and addressing the overdue need for novel solutions.

A product's life cycle is constantly in flux. It's normal to have additional feature requests, brand modifications, content adjustments and more. The systems driving these products are often unable to dissipate this amount of uncertainty and change properly.

Today's systems have been successful in their mission to a point. But they were not built with longevity in mind. While they once represented well manicured guidelines now they resemble a web of dependencies that have become increasingly difficult to straighten out. Leaving individual product contributors to fend for themselves.

As discussed previously, each stake-holder group requires certain information to be made available to them in order to work and collaborate effectively. The required information both shared and not resembles something like this:



Organizations are faced with the task of wrangling all these different information points into one standardized system. Their solutions, however, were not done in a way that allowed for future-proofing, shareability, or easy maintenance. Over time each group focused on their own documentation, ensuring their needs were met first. This left other stake-holder groups in a state

of uncertainty, trying to decipher eachothers documentation. The result of which is information duplication and a higher rate of “page rot” or documentation that has been neglected to the point of inaccuracy. As others come across these stale guidelines they take them as truth, further amplifying the issue. Organizations have attempted to remedy this issue by spinning up teams whose focus is the remediation of the forgotten documentation.

This is one of the reactionary responses that present day systems are unknowingly requiring of its contributors. It causes an unnecessary reallocation of resources resulting in bottlenecks in previously unrestricted areas of the team's workflows. Even if a team is successful in updating the documentation, if the cause behind why the documentation went unchecked is not assessed then it’s only addressing the symptoms of the problem. These patch-job solutions become a recurring event taking time away from the system's critical problems and drain resources from other parts of the application.

One external force working against the steady maintenance of a design system is the internet's intrinsic property of never staying the same. The development frameworks that serve as the code foundations for many digital products constantly change to keep pace with reality. Whether due to security concerns with outdated frameworks or incompatibilities with new versions of the code, sooner or later, teams have to make major changes to the product’s codebase. This is yet another example of a reactionary response to a predictable situation. For a product to be agile they need to to keep pace with this rapid shift in technology. An ability that requires a development foundation that is flexible. But, seen as unlikely and not worth the investment we once again find ourselves patching over the symptoms until the system starts to resemble a Ship of Theseus. The ease of saying, “Get there when we get there”, enables us to push important issues into the future until the last straw breaks the camel's back resulting in massively expensive rework. Whereas the upfront investment in a robust foundation would have negated many of these headaches in the first place.

We also find faults in the drive for product progress and growth. While perceived as growth, the expansion of product features, feature complexity, product offerings or team size, increases the relative complexity of each resource required to develop the product. This is a compounding effect as we extrapolate work overtime. Similar to how we may wait to the last minute to refactor dead code, by letting complexity get out of hand without an ability to functionality match that pace we inevitably start acquiring more problems. General employee turnover from requiring

teammates to work in unorganized and nebulous systems causes siloed knowledge to be lost, outdated documentation gets misinterpreted, and the original process for maintaining the system falls to the wayside as changes appear faster then they can be written down. Without proper care important information goes missing or becomes stale over time. Each previously mentioned stakeholder requires a lot of varying sets of information that range from process guidance to foundational product knowledge. With no reliable source of truth, teams spend time attempting to procure answers to their questions without assistance. Their daily tasks go from focusing on their work to trying to set up an environment where they can do it. Product information they thought was valid one day may differ from the next as information shifts around.

Unable to keep up, these systems become fractured into an array of different frameworks, styling, feature experiences, and information from disparate product knowledge in an attempt to keep pace with the world around it. Eventually this becomes too much to wade through and timelines begin to expand and progress slows.

As we've started to allude to, today's systems infrastructure and information management processes are not just a hindrance to the development of the product, but the proper cohesion between teams as well. The lack of communication and collaboration between designers, developers, and product individuals is a common theme. This lack of understanding leads to slow production times and a worse overall working experience for individual contributors. Scattered or lack of documentation on everything from expected experiences to the availability of resources within code leaves each group to fend for themselves. A rift forms between stakeholders as each spends most of their time trying to keep their ship afloat. This is where we start to get greater degradation in designer/developer collaboration and turnover and poor production quality become just some of the significant implications of this lack of interdisciplinary support.

While initially expedient, the popular approach to design systems soon reveals its shortcomings as maintenance costs soar and compatibility issues abound. The lack of cohesion and clarity within these systems impedes productivity and undermines digital products' integrity. The compounding nature of information requires a robust system to manage it all.

In the following chapters we will delve deep into the intricacies of the Agnostic Design System (ADS) and its counterpart, the Product Kit (PK), exploring the layers that comprise each. Luckily,

we won’t be deviating from many of the major concepts of today's system; instead, we will be optimizing them to work together. From the foundational principles of web components and accessibility to the nuances of collaboration and documentation, we will look at what it takes to enter a new era of system management for digital design and product development.

Chapter 3: The Universal Design System

We've identified various issues with today's implementation of design systems. Issues such as the fragmentation of information between collaborative stakeholders of a product and the lack of system adaptability which leads to instability and communication problems. Let's see how the Universal Design System can step in to solve these issues and improve the product design experience overall.

Today's systems lack the adaptability to efficiently pivot and stay on top of changing circumstances. More time is being spent maintaining the systems than growing and enhancing them. It becomes a job in and of itself not just for one team but all the teams involved. The root of which can be found at the core foundations underlying each segment of product development. We need systems in place that support the particular needs of our developers, designers, and product teams. A system that keeps pace and naturally steers itself towards organization without imposing restrictions on any other stakeholder involved.

To solve these issues we can intentionally arrange many of the artifacts in our existing systems following principles of the Universal Design System. While still maintaining many well known concepts of design systems like tokens, components, and more, we can reconfigure its pieces to allow them to work in a more harmonious way with one another.

When we look at a Universal Design System we will focus on two primary parts. Our underlying development foundation, which we will call our Agnostic Design System, makes up the first half. This is where we create an extensible and flexible component set underpinned by guidelines focused on ensuring we don't repeat past issues. The second half is a dual purpose concept called Product Kits. These aim to organize individual product principles and set up an avenue of communication between it, other products, and the Agnostic Design System.

Chapter 4: Agnostic Design System

The first part of the Universal Design System is the Agnostic Design System. This system is what enables the decoupling of the underlying development system from the products that utilize it. With this system we will be able to mitigate many of the issues that arise from the rapidly changing tech landscape and provide a solid foundation for a multitude of product applications to hit the ground running.

The agnostic system is more of a development framework than a design system. But the concepts we will be applying to it make it system-like, so the two are interchangeable. We will continue to call it an Agnostic Design System or ADS.

Development frameworks are used to provide the digital components an application needs to build out its features. Components like buttons, form inputs, error messages can all be pre-built and provided by these frameworks to improve development efficiency. But these frameworks often have embedded concepts that make them rigid. One example being that these frameworks will choose ahead of time what development system they are based on like React, Angular, Vue, or Svelte. Locking you into a certain ecosystem. These development frameworks also lack accessibility and other important features that enable scalability and customization. Once you choose a framework you are stuck with it until you are forced to spend the time and money to switch to a new one often due to a lack of support or feature set.

While company-built frameworks can be useful as they remove a lot of the overhead from the development process, the issue with them is that they often are based on pre-existing products who have embedded various product prescriptions within it. They may also not have certain features if the product never required them. Features like accessibility compliance which are often overlooked in the digital product world.

We don’t want to do away with the entire notion of a development framework. The core concepts and intention behind them are important. From reducing the overhead required to get product features stood up to making components reusable, these frameworks serve a very important role.. What we want to do is solve for their inflexibility and cut down on the friction these built in restrictions can impose on a product. All while keeping the system reusable and efficient.

The key to solving these issues can be found in the Agnostic Design System (ADS) methodology. The ADS is an agnostic development framework, meaning that it has no opinion or preconceived notions around its usage or implementation. More specifically, it has no imposed development framework or product specifications that often comes from product born frameworks. By removing these prescriptions and allowing implementers of the framework to have the freedom of choice we open the door to a more stable and enjoyable development experience.

An Agnostic Design System serves as the functional foundation for digital products and provides the components, behaviors, and any other development overhead you want to include. When constructing the ADS we don't want to make any decision that would require its implementers to become stuck in one spot or another. To the ADS there is no product that it is aware of. In a way, the Agnostic Design System is its own product and seeks to adhere to its goal of flexibility and efficiency.

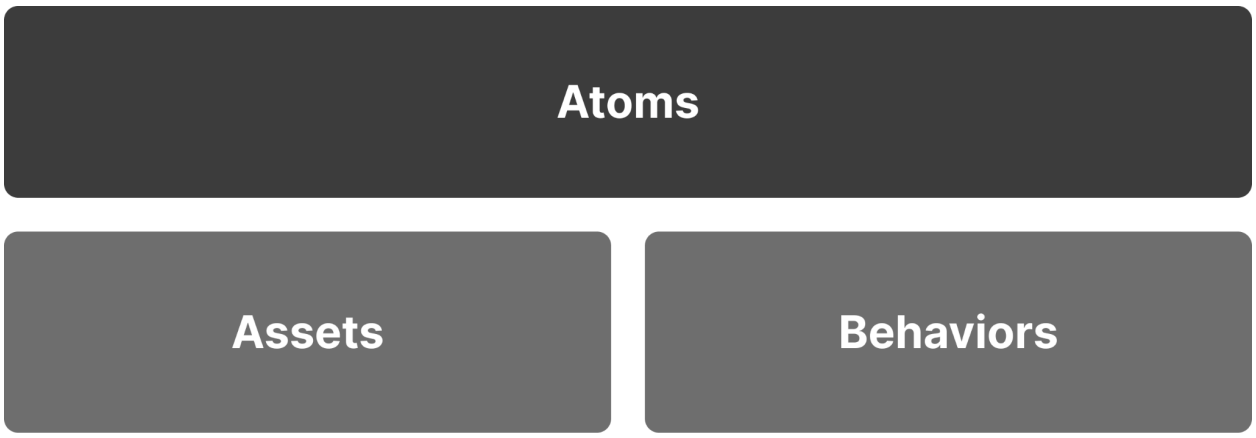
The ADS will be the foundation that products utilize to create their digital applications. This is done through the use of reusable functionality and components, as well as robust documentation. These three are what is required to produce a useful development framework and should all be built with the intention of being agnostic and flexible. Let's dive into the layers that make up the Agnostic Design System.

Layer 1: Assets and Behaviors

The first layer consists of assets and functional behaviors. This layer serves to primarily reduce the time it takes to stand up new digital applications. By providing a kit of assets, like glyphs and icons, we allow new and experimental applications to be developed at a quicker pace. This negates the need to curate graphical sets per application which can be costly when simply trying to do rapid iteration or create internal applications. This set should not be a requirement but rather an optional set that helps achieve the aforementioned goals.

Functional behaviors on the other hand are programmatic encapsulations of common human interaction conditions. Interactions like drag and drop and collapsing interfaces are examples of this.Instead of applying interactions uniquely to each component, we set them up as reusable code snippets. With a set of behaviors available for use the ADS can offer ready-made code to handle these interactions that can be applied to any component that needs it. This way we keep all the code referenced in one location, drastically reducing maintenance cost and complexity by

having one source of truth. Having reusable code like this also allows the behaviors to live on their own. Products who do not find exactly what they need are able to utilize these behaviors within their own component contexts instead of reinventing the same thing over and over again.



Layer 2: Atoms

With the next layer we get into the component structure of the Agnostic Design System. Components are the visuals we interact with and see within an interface. This layer is centered around a subtype of component called atoms. Atoms are the building blocks of the all preceding UI and are the most basic form of a component and usually do not contain any other components within them. The term “atom” comes from the atomic design concept where components are divided into categories based primarily on their simplicity and ability to be combined into more complex compositions. Buttons and input fields are good examples of these as they are basic but can be used in conjunction with other components to produce more complicated UI, like forms or navigational components.

When forming our layer of atom components we want to take care to ensure we are ensuring flexibility and customizability along with the other core principles of the ADS. This is where the

restrictions of previous development frameworks come in. Company created frameworks often prescribe systems like React or Svelte to their component sets, requiring the continued use of these systems for the rest of the product's life. Where the only way around it is a costly conversion process. To get around this prescription we can utilize a powerful web standard known as web components that will enable compatibility with all the development systems that are out there.

Before we had many of the popular development frameworks we know today the web was built with a combination of HTML (hypertext markup language), CSS (cascading style sheet), and a touch of Javascript to enable interactivity. Buttons were coded using syntax like <button> and provided a lean representation of what a button was. Whereas today's frameworks wrap this lean and efficient solution up in extra layers that abstract its core purpose away and fill it with bloat from added features. Web components try to remain true to the original HTML definition while exposing key features to enable customization. This is a perfect solution for the Agnostic Design System. We can utilize this lean base as the foundation of our components code. It may sound like we are just going to be adding our own bloat to the system but that is not the case. The key is to not take things too far and extend things just enough to achieve the goals of the ADS.

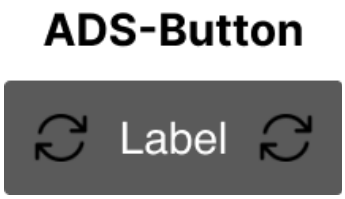
We want to avoid what other product frameworks have done. Where they are adding layers to the core HTML so they can interact with existing systems like React and perform hyper-specific functions that can pigeonhole its implementers. Any and all product definitions must be taken off of the framework. How a component looks, what icons it has, or what it is used for is not up to the framework. The onus is on products implementing the framework within their application to decide those details. This is where the Agnostic Design System gets its agnostic trait. The ADS is able to avoid over-prescription and enable those prescriptions to be applied via the product. Enabling an efficient, flexible, and lean route to execution.

With the utilization of web components in order to create component foundations that are agnostic and unrestricted we want to ensure their ability to remain customizable. Today's frameworks often come with pre-styled components. While helpful to a degree, this falls apart once anyone else wants to utilize it for a new application. The inclusion of these extra styling muddies the waters and is essentially useless for any other application as their look and feel will

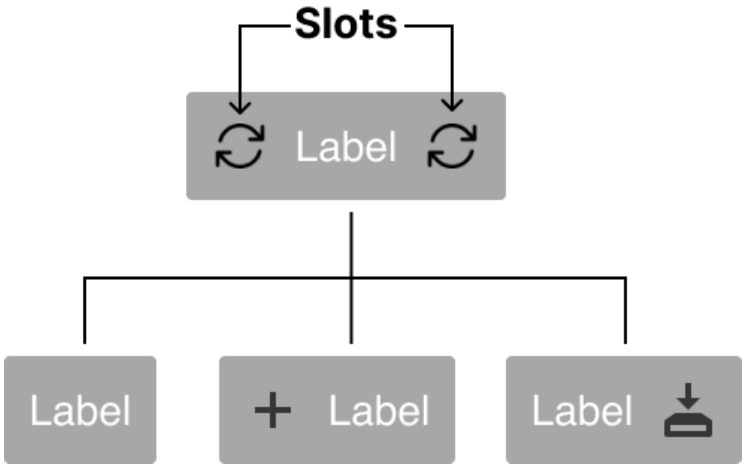
vary dramatically. The ADS wants to avoid this stylistic prescription for its components to ensure that they can be customized in the future.

To achieve this we want to expose all of the styles within a component for implementers to utilize. Even considering the inclusion of marked locations for additional components and assets to be placed. The goal is to provide all the levers to a component but not set the positions of them, that is up to the products who use the ADS.

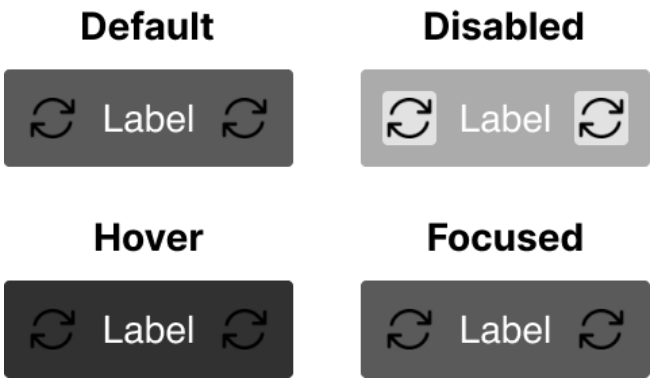
Let’s look at this through the lens of the button atom component. Buttons come in many forms and are intended for a wide array of situations. Defining the core ideals of buttons is important to creating its agnostic version. Buttons are clickable elements that perform an action. They may be square, circular, gray, blue, or even an icon. At the ADS level we don’t know how it’s going to look so we want to leave it open to interpretation. While we could just provide a blank UI with functionality to handle a click interaction that wouldn't be terribly helpful. Along with that functionality we want to provide avenues to customization.



This is where we get into slots and tokens, the customization structure that allows for ADS implementers to apply their visual specifications. Slots are placeholders within the component to allow for the inclusion of other components or assets within it. In the case of the button we can include slots on the left and right of the buttons label, providing locations for icons to be placed.



Having slots can massively improve the speed in which teams can implement the components on the product side. Slot locations can be set up to cover a large swath of component use cases while remaining an optional feature for implementers.



As with all components in the ADS, we also want our component atoms to come from some core states. For the button this might be the inclusion of a hover, focus, default, and disabled state. Having these states makes the application of styling more straightforward and paves the way for more robust accessibility adherence as different states require different accessibility features.

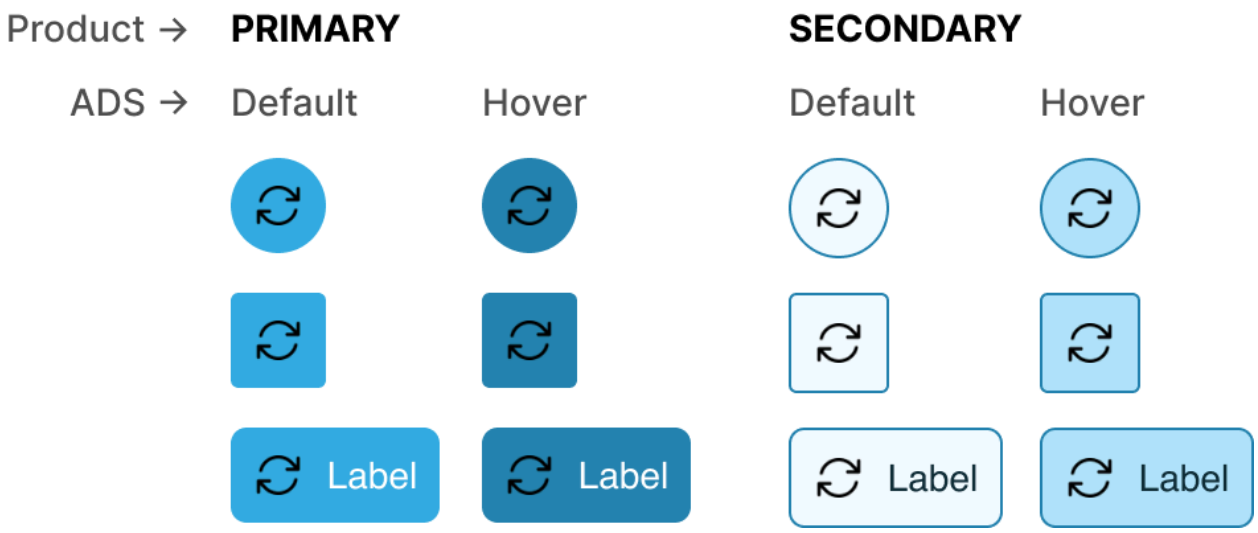
Button Style Specs

stroke
slot-fill
background
border-radius
etc.

padding-left
padding-right
padding-top
padding-bottom

With these states we add some organization to the customization of the component. Within these we can also provide access to the components various style properties. We can expose this customization in a structured way through the use of CSS templates.

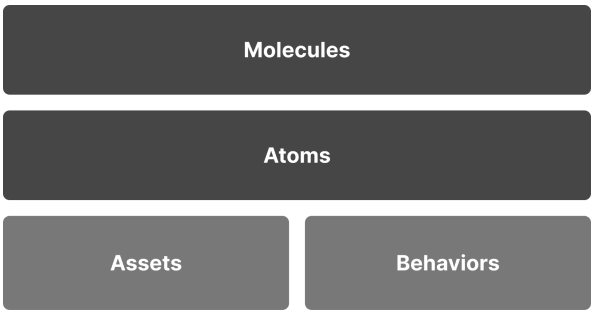
Components come with CSS that implementers can utilize to apply their specific styling to. This can get laid out with states as the high level groups and the parts of the components as the children within them. Each state also has spots for the customization of strokes, backgrounds, spacing, and more.

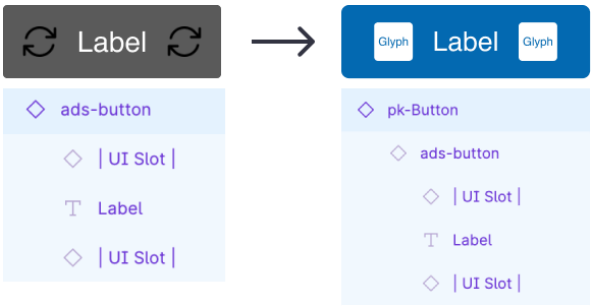


Today's systems often come with concepts like primary, secondary, and tertiary for their buttons. These are product specific concepts that we avoid prescribing with the ADS. The ADS simply supplies access to the styles and allows products to decide what they need.

Layer 3: Molecules

The next layer of the ADS is molecules. These are very similar to atoms but are more complex in their construction and purpose. Where atoms are basic components that represent well known UI like buttons and input, molecules are compositions of atoms and behaviors that are geared towards more specific circumstances. These could be headers, alert messaging, dialogs, or banners. Essentially any combination of atoms and molecules that form a new component. Molecules primarily serve as a categorization to separate simple from more complex components to help implementers make better sense of the system as a whole. At the atom layer you could expect to have much more freedom in terms of composition then you would at the molecule or higher layers.

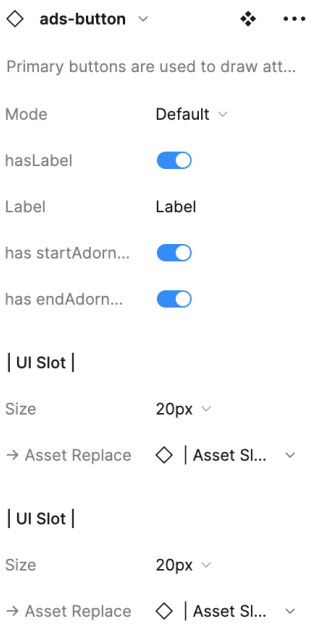




It is important to remember that frameworks are not just for developers. In order to maintain healthy collaborative relationships with other stakeholders the framework needs artifacts that allow for this communication to take place. To achieve this we can include a design library based on the framework for designers to work from and developers to reference. This is a very important artifact as it helps bridge the communication gap between these two groups. Product teams will pull the ADS design library into their product specific design library and use its components as their foundation. The ADS library is not used directly, it is brought into new libraries and styled against product specifications. This way we retain naming, slots, and more that help communicate the construction of components to design and product stakeholders. When we look at a component within the Product Kit we will be able to see what components from the ADS are in use and how. Allowing developers to get a clearer picture of how the components are built.

Here we see the Agnostic Design System representation of a notification item and its transformation into a Product Kit notification. We want to ensure that the design kit that designers and developers are working from is as flexible as the system it represents. The ADS notification is simple, unstyled, and customizable. Slots are used to show where other UI, like atoms or text, can be slotted in. While slots are optional, they are a no cost way of getting implementers 90% of the way there without hindering their creative ability. The goal of the ADS

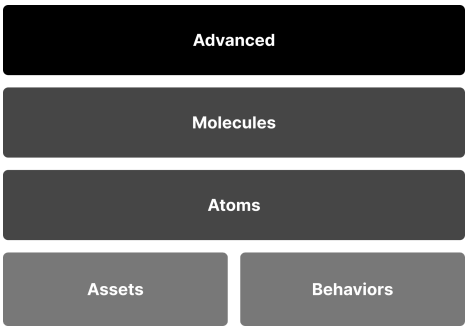
is to provide that foundation of pre-done work to reduce the overhead required to set up that foundation. Once we set up that foundation, we negate the need to do it again for every single product that is going to use the ADS as its underlying component architecture.



In this example we can see how the properties panel in an application like Figma would look for the button component. Clearly outlined are the different modes or states, locations for slots and designer experience enhancers like hiding and showing certain content. Consider adding additional tooling to the design kit for things like slots. In this image we can see the “UI Slot” that has been constructed specifically to handle either the swapping in of full components or the insertion of assets, like glyphs. The swapping mechanism has been surfaced for designers within the Product Kits to quickly customize the components to match their visual specifications.

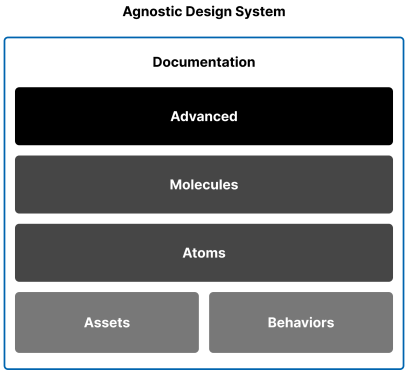
Layer 4: Advanced

Layer 4 is the last component related layer in the Agnostic Design System. It primarily serves to categorize the most complex and massive of components. Things like tables are a good example of this as those are very large UI composed of many atoms, molecules, and advanced behaviors. The same rules apply where we want to leave things open ended but set up the framework for others to work from.



Layer 5: Documentation

Documentation wraps everything together and is used to explain the use and implementation of the Agnostic Design System. Here you want to ensure definitions are in place for all of the available components within the foundational system. This resource should also be set up to serve the



other stakeholders as well. Utilize a documentation system, like Storybook, to help publicize this content to your stakeholders. This documentation should lay out some key pieces of information. Outline the styling API so consumers understand how they can go about customizing the UI with CSS. Provide the various properties a component might have like states or functionality options. Consider providing some extra documentation such as interactive versions of the components so designers can embed them in their documentation and product folk can understand the capabilities of the UI. Have a theme selector so products can see how components look for each of the applications. Even consider including accessibility checkers to showcase the components compliance with various WCAG guidelines. It's important to get this documentation done right as it is going to serve many different types of individuals working on the system. All of which have their own needs and goals in mind. Not only that but instructions have to be clear enough to explain how individuals go from the unstyled and basic ADS component to their own Product Kit component.

Chapter 5: Product Kits

Where the agnostic system provides us with the foundation of core componentry that comes with built in accessibility, customization, and functionality. The Product Kit serves as the product governance that turns the ADS into usable product experiences. They represent the coded product specific components and the kit that designers use to piece together the product experiences within their design application.

We can think of Product Kits in a similar way to regular design systems and are essentially the fingerprints for products. They contain very specific visual and user experience guidelines that dictate how components get composed. Content like type, color, and spacing scales all get defined here. The Product Kit will then apply these standards to the ADS foundation to create the product specific components.

Product Kits (PKs) are specifically separated from the underlying ADS to allow for the ADS to support any number of products and not be hindered by the requirements of any one product. If the Agnostic Design System made visual and user experience prescriptions to all those who used it; then there would be an increased risk of timeline and objective collisions between the products and the ADS. Since every product would want their work implemented in the ADS to support their own goals, the ADS would have to pick sides and prioritize work arbitrarily. By setting up Product Kits, we encapsulate this responsibility to the relevant parties and allow them to work as fast or slow as they desire. All in all, the ADS is responsible for the reusable foundation and PKs are responsible for the product.

All the core concepts of Product Kits that we will be covering are reproducible without the use of an agnostic component foundation as they focus on creating a managed and scalable product design system. You will be able to disconnect it from an underlying ADS but the ADS allows them to work to their full potential and scale in a far more efficient way. Since the PK's are using a foundation of flexible components the aforementioned issues of overhead and shifting technologies are greatly limited. The concepts we will focus on are documentation, components, and theming.

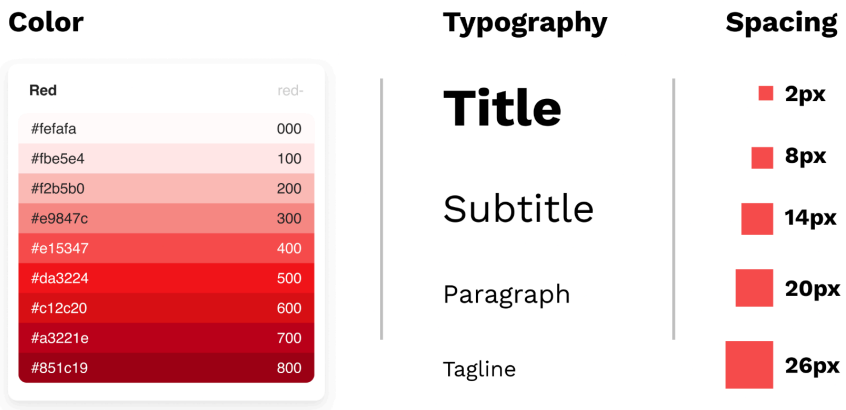
Product Theme

The theme of a product is the visual logic that describes how components for a given product look to the end user. Such as prescribing all alert notifications the same red colors or all primary buttons utilizing the brand's main color. This consistent ruleset for visuals allows for stable adherence and consistent communication between teams. Having a well-thought-out and systematic theme allows teams to think less about what styles go where and focus on the product instead. Themes can be

extended with additional logic to allow for sub-branding, white-labeling, and alternate product modes. Modes like light, dark, and high contrast.

Themes in a PK are also set up to form a bridge between the design and the code. As we will discuss, the usage of tokens allows us to have a direct connection with the styles within the developed components by replacing their static values with adjustable variables. This is very important as it brings more of the responsibilities to the correct groups. Developers get to focus on the code and functionality. While designers are able to ensure consistency with a single source of stylistic truth.

The bulk of a theme consists of all the well known topics in regular design systems. When we look at a product we see a defined set of styles that make up the entire application. Spacing between elements used, colors for components, type scales like headers and paragraph text, all of which is being used in a prescribed and intentional way. This information can come either from the existing product or from scratch, it all depends where you are starting.



It’s important when creating a PK to have these low level style definitions created so you can quickly set up the visual foundation as it will inform the rest of the pieces going forward. Consider including not just type, spacing, and color, but also any drop shadows, gradients, and borders, if they are used within the application.

In order to get all this information into the Product Kit we want to morph them into a theme based on data and logical organization. When we think about themes we are primarily looking at how we can use tokens to tell the visual story of our product components in a way that

translates well for designers and developers. So it is important to implement a proper token structure for the entire product component set.

Let’s dive into what tokens are and how they work together to define our product visuals. Feel free to follow along in a design app, like Figma, that allows you to set up token structures.

These are not to be confused with the more programmatic definitions of tokens like API or authentication tokens. Nor should you mistake them for arcade tokens or bus fare. Many refer to them as design tokens but we well simply refer to them as tokens.

red-500 = #da3224 

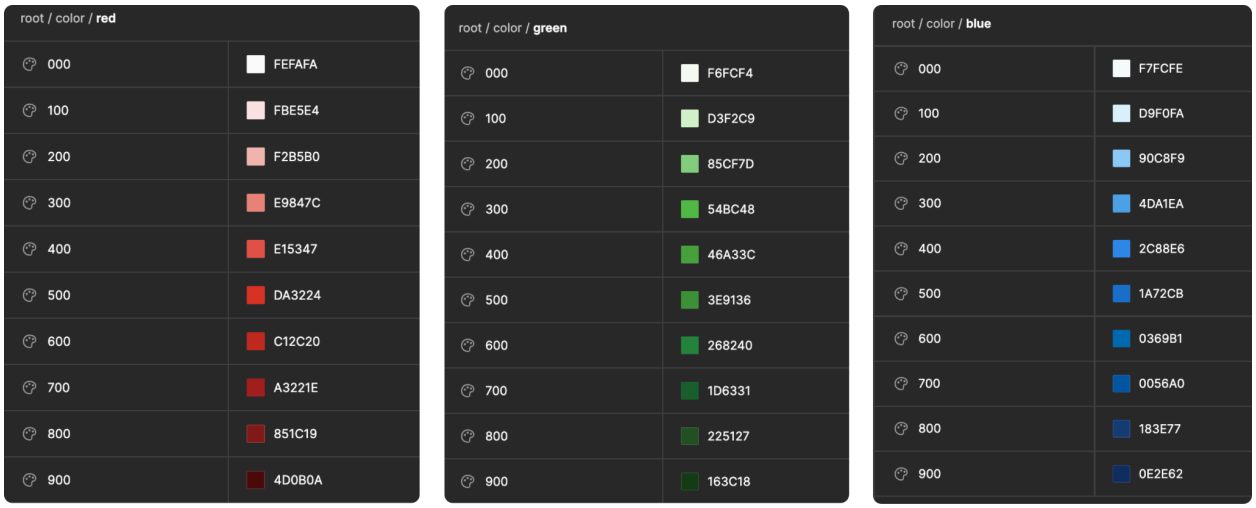
Tokens are key:value pairs that outline and orchestrate the complex nature of component styling. Where we use the key as the new name we want for a value. They describe every style defined by the product and often come from existing brand guidelines. Tokens are inherently reusable and through the right orchestration allow for dynamic relationships between components and their styling.

We define the theme tokens with three layers, root, alias, and component.

Root tokens are the foundational style layer that the other layers work off of and is where we will be taking the aforementioned style scales and outlining them for the rest of the layers to work from. For example you may have a set of reds, greens, blues, and magentas that are all tuned to the product's identity. These get defined as root tokens in a very plain and simple way, with the goal of turning our unintelligible hex codes into readable text.

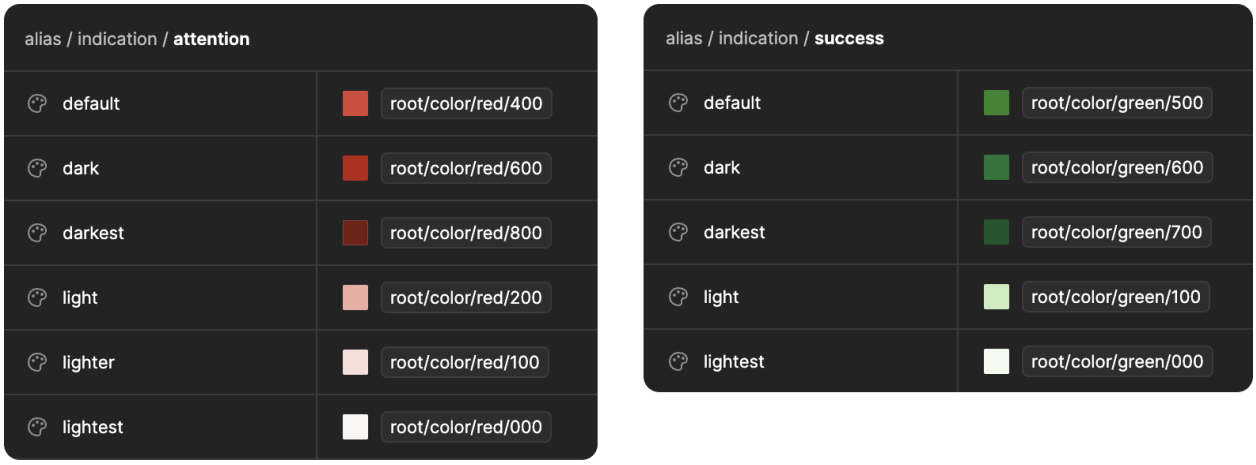
Your red color scale may look like this, red-100 = #eb4034, red-200 = #d1291d, etc. Where you are providing understandable names to the hex value that represents the real color. It is recommended to follow this naming scheme of color-### as we want the root layer to impart no notion of expected usage upon the colors. Using a numerical scale allows us to just portray color characteristics. Where we know red-100 represents a light red color and red-900 represents a dark red while allowing for new colors to be added at the end or in between with

values like red-150 or red-1000. The root layer simply outlines what styles we have at our disposal and when defined for all our styles afford us with a set of reusable styles.



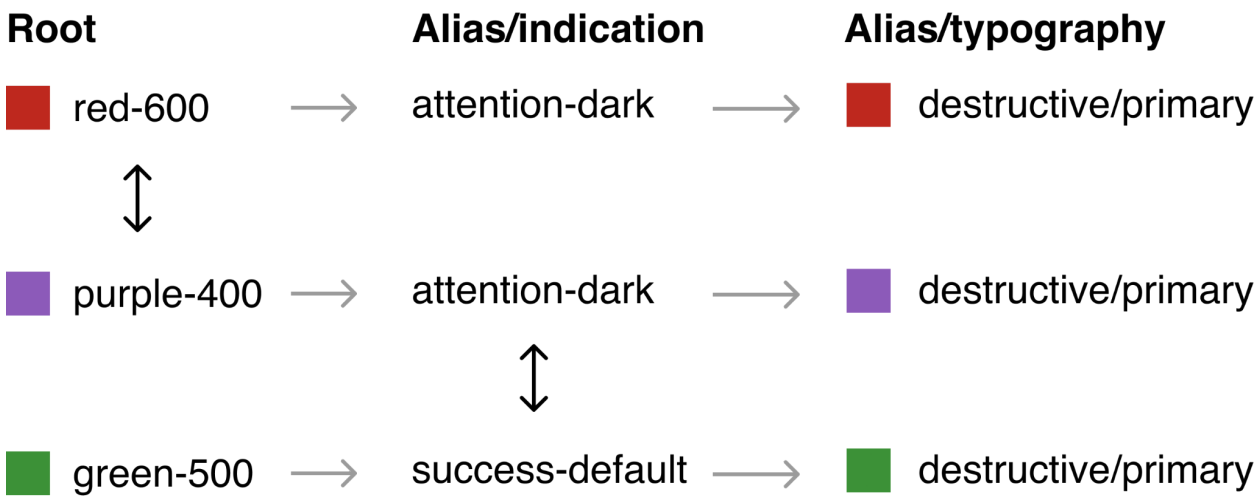
Now that we have our product styles defined as root tokens we can move on to the next layer, alias tokens. Much of the dynamic nature of themes and their tokens comes from this alias layer. This is the middle layer between root tokens and component tokens. Aliases get categorized into groupings based on design standards that can be reused for various components. Their names are more specific to their intended usage. Categories could be branding, notifications, alerts, containers, etc. These categories can be made from scratch but if you have existing components it is much easier to go through each component to find the similarities between them to form a starting point for your aliases. Ask yourself, “What styles am I seeing being reused?” and “Do these reuses have anything in common?”.

Given a situation where the product wants to use the same set of red colors for their error treatments and green colors for success treatments, aliases pave the way for what that definition is. Gather up the root tokens you want to use for alert and success components and create a new naming convention to house them. A good name for this could be “indication” with child groupings within it called “success” and “attention”.



Now, where root token names might look like red-400 = #E15347, the alias token would be a layer or two deeper, indication-attention-default = red-400. The “indication” keyword is our parent grouping that will house different forms of indication types. Types like the next keyword “attention”, but this can also be success, warning, or information. The last keyword, “default”, indicated the level of color for that type. We still want our aliases to be somewhat vague as the actual use of color might differ between strokes or backgrounds of particular components. If you find yourself using the same attention alias token often then that is a good sign that another set of aliases should be set up for that use case. It all depends on the level of prescription you are aiming for.

All we are doing here is partitioning off a select group of styles so our tokens and components further down the chain are referencing the same set of alias tokens. This will enable both style adherence and easy re-styling as the components will be all pulling from the same set. When you change any of these alias tokens then the downstream tokens and components will update themselves without additional intervention.

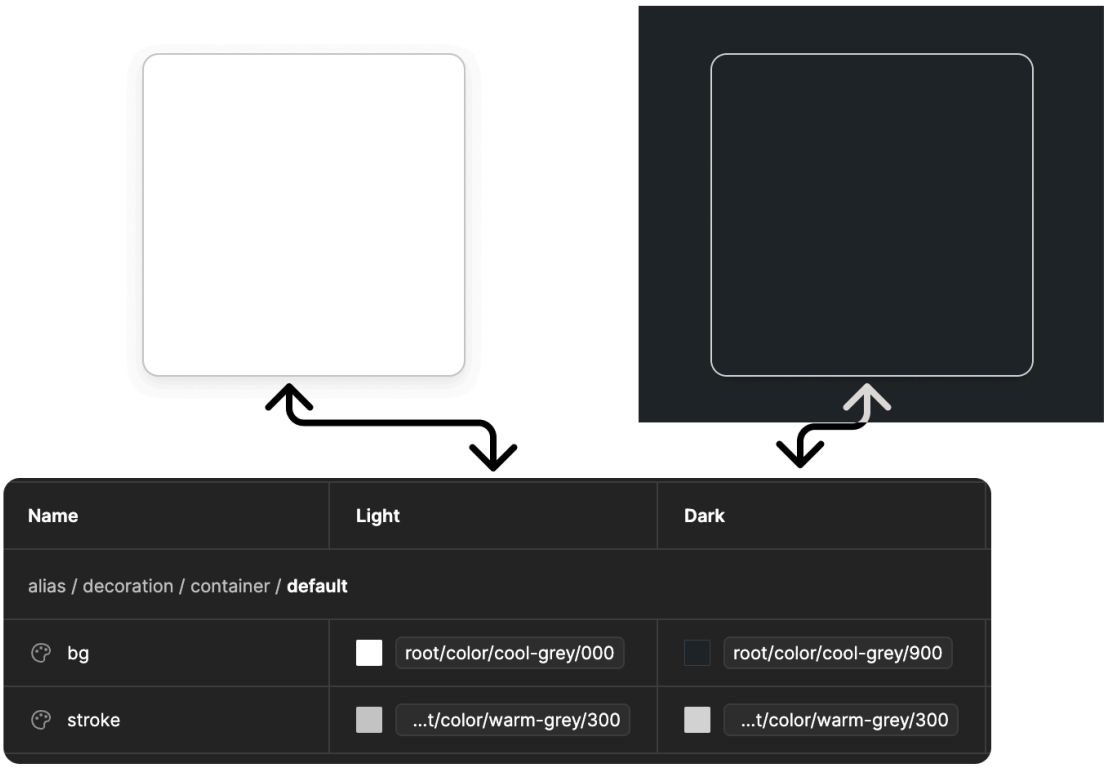


Here we are first reassigning the root token that defines indication-attention-dark token. This causes all downstream tokens using the attention-dark alias to change as well, causing the typography/destructive/primary alias to change. We are not relegated to only swapping root tokens around. Any token that is being defined by another token can have that token changed in order to be re-styled. Like where we are changing the attention-dark to success-default, causing the destructive-primary alias to change along with it.

We can take this a step further by looking at re-theming or sub-branding. Since we now have groups of reusable aliases we are able to have a sense of certainty around color usage for all our components. Assuming you utilize only aliases when applying style to your components we know what will be affected by token re-assignments. This is useful because when we want to incorporate something like dark mode we don't want to have to go into every component and change each individual value. As well, we want to ensure consistency in the new color usage. It is highly unlikely that in this new theme you will have varying colors for primary destructive typography. If you did that then the brand would be lost as you are changing around the visual expectations of the product. This is not to say that your application couldn't have different forms of primary destructive text. It is not unheard of to have different colors for different typography sets like headers and paragraphs. But that stylistic concept is also likely already a part of the original application theme so they would have been set up as aliases prior to creating a dark mode theme.

When theming we are essentially setting up a second column of token assignments for our aliases. Re-assigning what root styles an alias originally was pulling from to create this new

theme. We still keep our foundational root layer intact and untouched as this base layer should already have all the colors you need to create a new theme. One column represents something like light mode and the other dark mode. Swapping one column for another causes that cascading effect that changes all downstream styles to match the new definitions.



In this example we have our container alias token set that we use for all instances of a container like UI. In the light mode the background and stroke alias tokens are set to light colors whereas in the dark mode they have been re-assigned to dark mode friendly root color tokens.

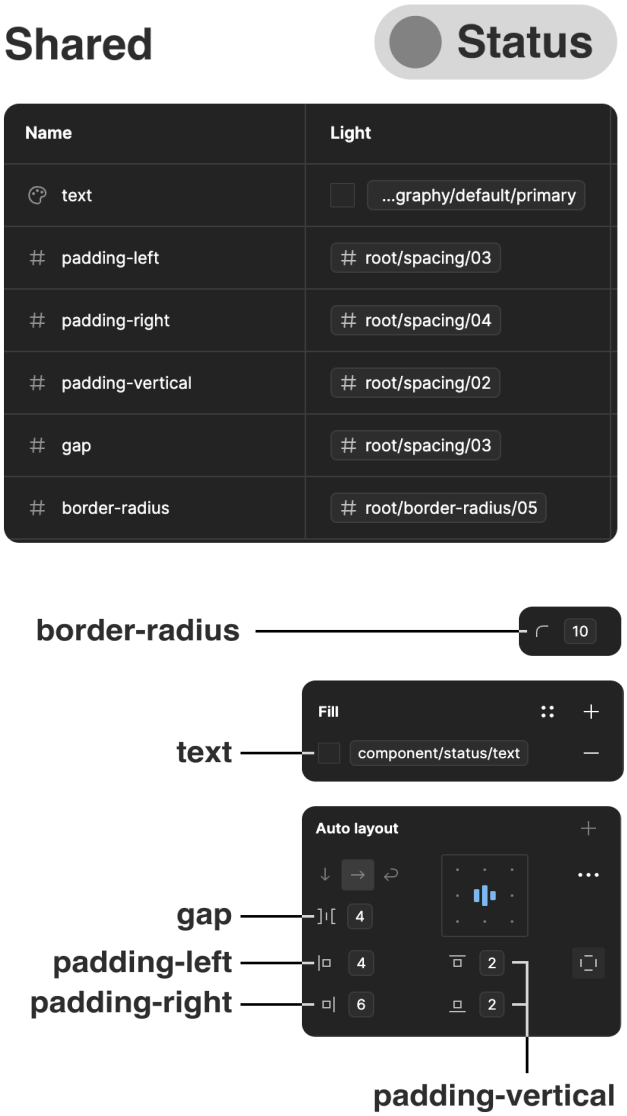
With our alias tokens we now have a set of global, reusable, and adjustable definitions for all our like-minded styles. These two layers, root and alias, are a very powerful combination that will reduce the time and effort required to make adjustments to your suite of components.

Many stop at alias tokens, but we are going one layer deeper to component tokens. Component tokens are the highest level of style token and intended to be a verbose representation of your components. Each component that you have will have a token for every single style that you

have applied to it. Every space, color, gap, margin, and so on, will be documented within this layer of token. Doing so allows us to have some extra fine tuning around how our components get styled, it creates a clear schematic for developers to easily understand what tokens go where, and abstracts the token assignments out enough to make token swaps easier in the future.

For every component you are going to have two primary categories of component token. A set of shared tokens and a set of specific, type based tokens. The shared tokens are styles that are intended to be shared with all variations of a component. While the specific tokens are meant to describe the differences in style between variations of the same component.

Let's start with the shared tokens on our example status component. With this component we will want to first look at all of our status variations and pull out the commonalities. In this instance we will create tokens for the padding around the icon and text, the gap between those two, and the border radius. We will also be creating a shared token for the text. In this example we want all variations of the status to share the same text color and by creating just one token for them all to use will accomplish that.

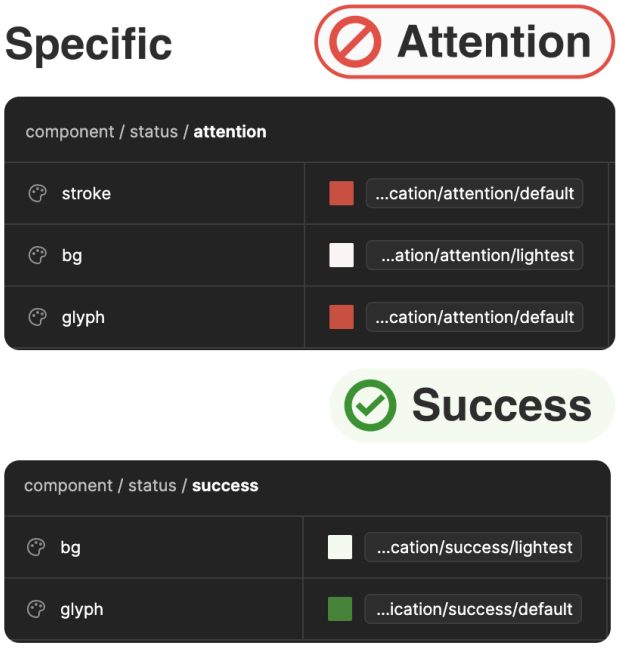


Before we dive into the specific tokens, which will be what styles our other status variants like success and attention, let's see how we apply these to our components. Component tokens are the last stop in the token hierarchy and are the values you will apply directly to your components. In your design application take the component tokens you have created and attach

them to the styling properties of that component.

Here we can see where each of the aforementioned tokens would go. Their naming scheme lends them to be easily identifiable both when applying them and when reading them after the fact. We know that the padding-left and padding-right will both get applied to the padding properties. This worked for the rest of the styles as well where the gap gets placed in the gap property and the border radius in the radius property. When it comes to reading the styles after the fact we also see that after application the properties are now populated with the token name. Developers are able to look at your component styles and see, for example, the text color is using the component-status-text property. This is rendered differently in the design application as “component/status/text” but that will map to “component-status-text” after you export your tokens to code through JSON. As well, for a design application like Figma Dev Mode converts these names into CSS for you so there are many ways for developers to get the information they need.

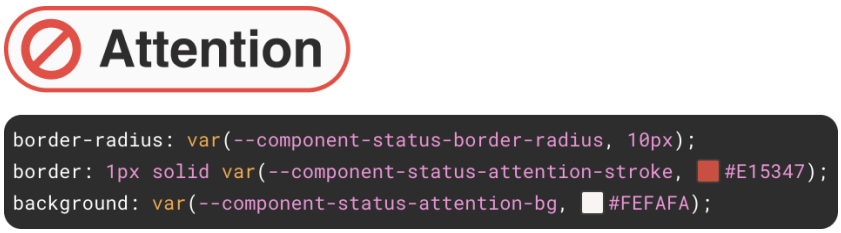
Now let’s look at the specific values. We have our status styles templated with the shared values but we need to supply the style definitions that make our statuses look like attention or success statuses. This is done by creating groups for each of our variations and defining only the differing styles between them and the shared styles. In our case this will be the background, asset color, and stroke color.

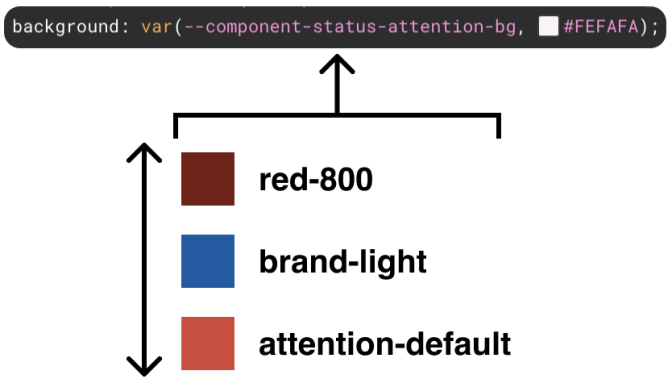


For the attention and success status variants we utilize the indication alias tokens we already have set up to supply the reds and greens for its coloring. In the case of the attention status we are also defining a stroke whereas the success status we are not. The visual guidelines say that success statuses do not have a stroke where the attention status does. By adding a stroke token for the attention status it overrides the fact that our shared

style status did not have a stroke. With the success status we don't add a stroke so it inherits the traits of the shared status, which does not have a stroke.

Now we are able to supply a full scale of tokens for our components from roots all the way up through component tokens. While many stop at the alias level it is highly recommended to continue up though that last component layer. It makes the components easily readable for anyone that comes across it, like developers and designers. The names are very specific and are written in a way that describes the exact piece of the component it is being applied to. Component tokens also allow for minor tweaks if your alias groupings are not cutting it. In cases where you have a one off color change for something like the attention status you don't want to try and add that to the aliases because it is unique to this one instance. Instead you are able to assign that definition to just that component through component tokens as they are unshared and decoupled so they will not affect anything else. Component tokens also allow you to take your time with styling. Making all these tokens takes time, especially the alias groups. Noone can tell the future and that being the case the alias groups will be forever changing. As well, a big no no is the application of root tokens directly to components. Doing this makes for a very rigid styling structure and negates all the benefits of the alias layer. It makes it so, in code, we have to go and edit the CSS because the name of the token has changed. When the components get coded the token found in their properties panel will be what is applied to the coded styling. If you are using roots or aliases on a component and change what token a component uses then that change will have to be reflected in code. By having the component token layer we are able to use that as the unchanging value that connects your designs to code. We will always have a value to refer back to in code so the design application can just reassign what is linked to that value and the





code will get updated without having to rewrite code. This way you can define your component token however you need without compromising the coded system. To make this even better, you can also include more component tokens then you need at the time. Even if your component does not have a color for a stroke or spacing in certain areas, including component tokens that are assigned to nothing lets you come back in the future and supply styles if needed. This way you already have the component tokens in place in code so the reference is available and won't need to be added in by the developers.

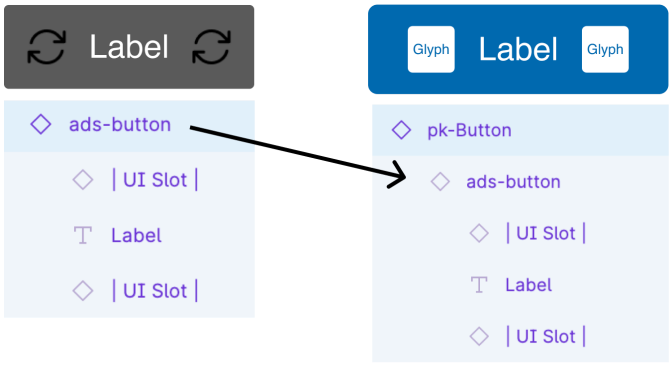
With our three layers of tokens describing all of our visual standards for our components we have completed our theme for the Product Kit. The theme will primarily be the responsibility of the designers since our component tokens will allow them to simply supply updated token references in order to make new stylistic changes to components. A simple swap of a JSON style file is all that is needed and no longer do you need to have back and forths between teams trying to simply update the stroke color of your buttons. This way we are able to bring relevant responsibilities closer to their owners and will play a part as we move into our next topic, Product Kit components.

Product Components

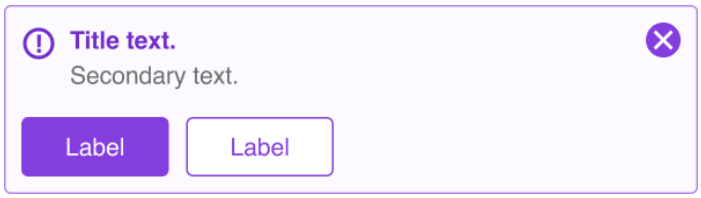
We have touched on components a bit in the previous section on tokens. But we have to get to the point where we can actually apply the tokens to something. Components in a Product Kit can come about in three main ways. They are either entirely constructed from ADS components, are a hybrid of ADS and Product Kit components, or are solely Product Kits components. PK components are components that are developed by that product and are intended for that product.

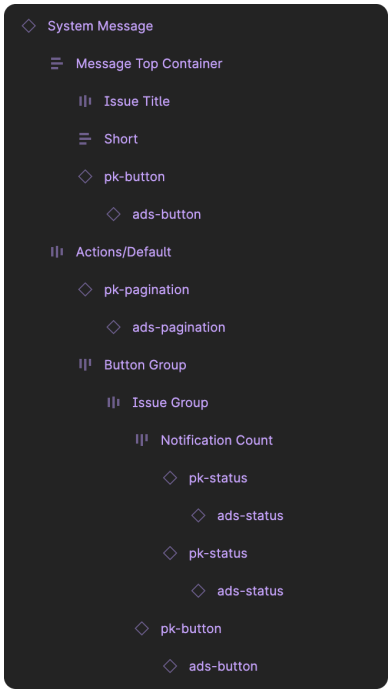
ADS only components are essentially theme only components. These are taken straight from the agnostic system and are styled by applying Product Kit themes to them. They do not require additional code, functionality, or composition as they come ready to go out of the box. The vast majority of components will likely fall under this category since atoms make up much of this layer. Functionally, compositionally, and stylistically atoms are very straight forward. Buttons, inputs, and statuses all fall under this category. They don't do much and what makes them different from one another is purely styling. So Product Kits don't have to do much more than style them with their themes.

On the development side this is just CSS theming using the tokens from the design application. But the Product Kit within the design application has to clearly outline the schematics for developers to know how the component came about. This is where we will want to ensure we have a base library for the ADS that gets used within the PK design library. The ADS design library serves as the reference point and base layer for the Product Kit components. When you are creating your components, start by pulling over an instance of that component from the ADS library and turn it into a new component so designers are able to consume it in their designs. Ensure that you are also applying all the theme styles to these new components as well. This way designers and developers are able to see both the theme structure and the construction hierarchy of all the Product Kit components. Components then take on a very verbose layering system that clearly shows what parts of a component are direct from the ADS and what are custom to the Product Kit.

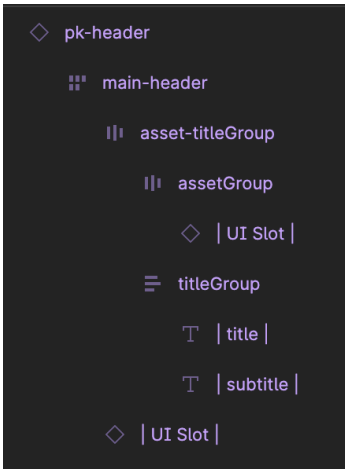


Hybrid components are a middle ground between fully custom product components and out of the box ADS components. Product Kits can take parts of the ADS component offering and combine them into new compositions to fit specific user experience requirements. This situation is as likely to happen as the utilization of styled ADS components. These might be cards with content within them or notifications. The agnostic system might not have the exact card component a product requires. But through the combination of a couple styled ADS buttons, a styled ADS status, and some extra styles, products are able to accomplish everything they would need with much less overhead. In a system that lacks the shared ADS foundation, every product looking to use buttons or statuses would have to construct them from the ground up each time. Looking at this Product Kit specific system message component we can see that it is not entirely built by the Product Kit. While the component as a whole is solving an issue designated by a single product, the ADS is able to provide the majority of the pieces that make





up the system message component. The layers show that, under the hood, the PK has rethemed the button, pagination, and status ADS components then combined them together with some extra styling. When it comes to the application of tokens we only need to concern ourselves with the ones we have not yet made. This component in particular is using many previously established components. For instance the primary and secondary buttons should have already been themed. The system message simply uses them again so there is no need to create new component tokens for them as they already have component tokens. The only tokens the component will need are for styles that are not given to it from another component. Styles like the background, stroke, padding, and border radius of the container. However you could create component tokens for something like the button if, in this very specific instance, you wanted to have a button style that does not match any of the existing button styling options. The flexibility of component tokens lets you perform this fine tuning since we are not beholden to any alias grouping standards we have created.



The creation of solely Product Kit components is for situations that the Agnostic Design System does not cover. They are coded by the Product Kit team for the Product Kit. This starts to touch on some governance processes around a UDS that define when products should deviate from what is available in the agnostic system. The purpose of the ADS is to have a foundation of common components so the case may arise when it doesn't have something a product needs. The team that owns the ADS could add it to the foundation but that is only recommended if there is a need by other Product Kits. Otherwise it poses a threat to the timeline of the ADS and could set work back if they are trying to accommodate everyone's one off components. Since the ADS

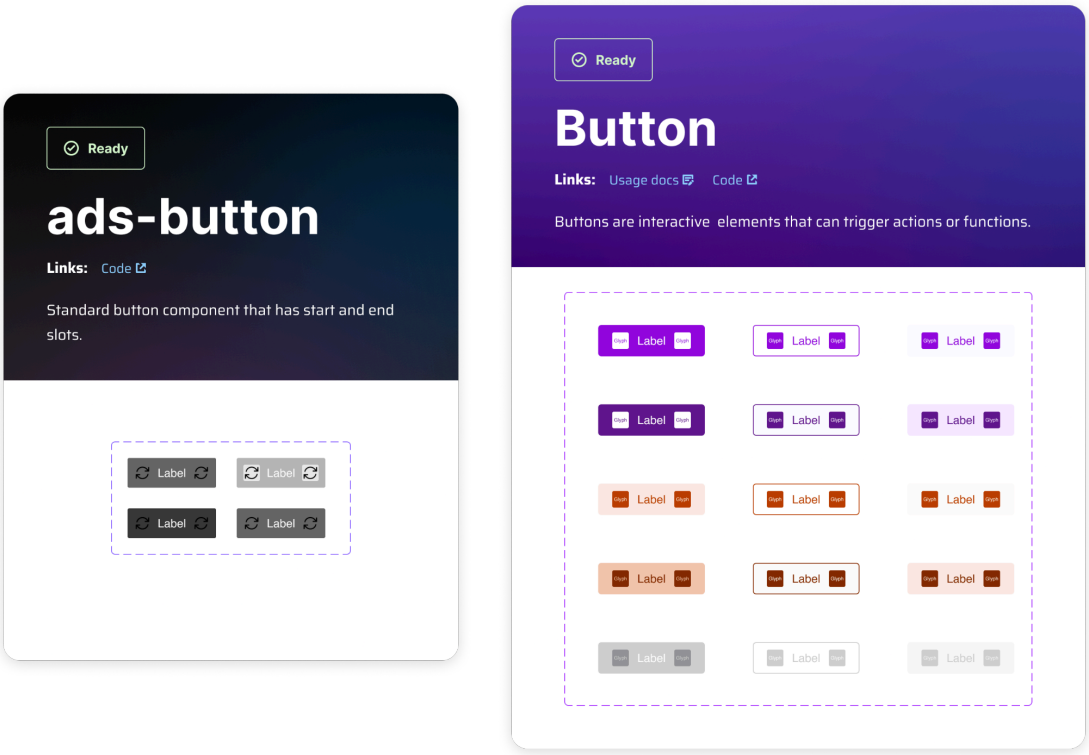
and PK's do not rely on each other, teams are given the option to work at their own pace, preventing this collision of priorities. In our header example here the Product Kit team and ADS team may determine that headers are a component that shouldn't get included in the agnostic system. The responsibility is now on the PK team to create and implement this component in their space only and we can see this reality reflected in the components construction. Nowhere in the layers are references to ADS components. The only items in there that could become ADS components in the future are the slot locations. These are intended to be replaced with various components depending on the circumstances. Some product features might need buttons on the right or a dropdown arrow on the left. These slots allow for that customization to be in place and are seen across component types.

Product Documentation

Product Kit documentation differs from the Agnostic Design System documentation in some major ways. Where the purpose of ADS documentation is to provide accurate explanations of how one would go about using its components and behaviors, the PK documentation serves to showcase the interaction model of components and experiences in the product. PK documentation is made up of repeatable experience flows that allow designers and product focused individuals to keep the product experience consistent. It offers answers to questions like, "When do we use a primary button over a secondary one?", "What is the user experience for editing list items?", and "How do errors work in forms?".

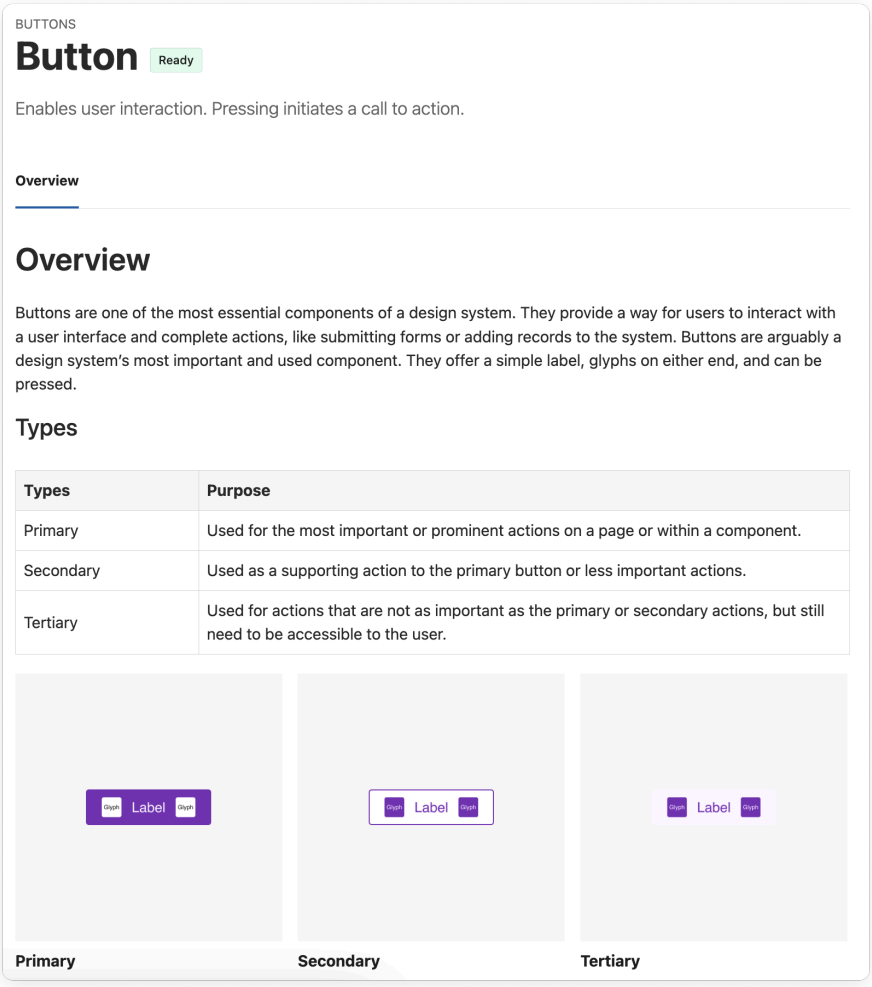
The goal is to provide all three stakeholders, designers, developers and product with the means to properly identify all the details they need to know about the available components and experiences in the product. Where developers' needs may end at token availability and the product needs end at what components are available for use, designers need direction on how to construct feature experiences.

Since the needs of Product Kit documentation range so much and the capabilities of most documentation services are not all encompassing, we will likely have two places where the information is housed.



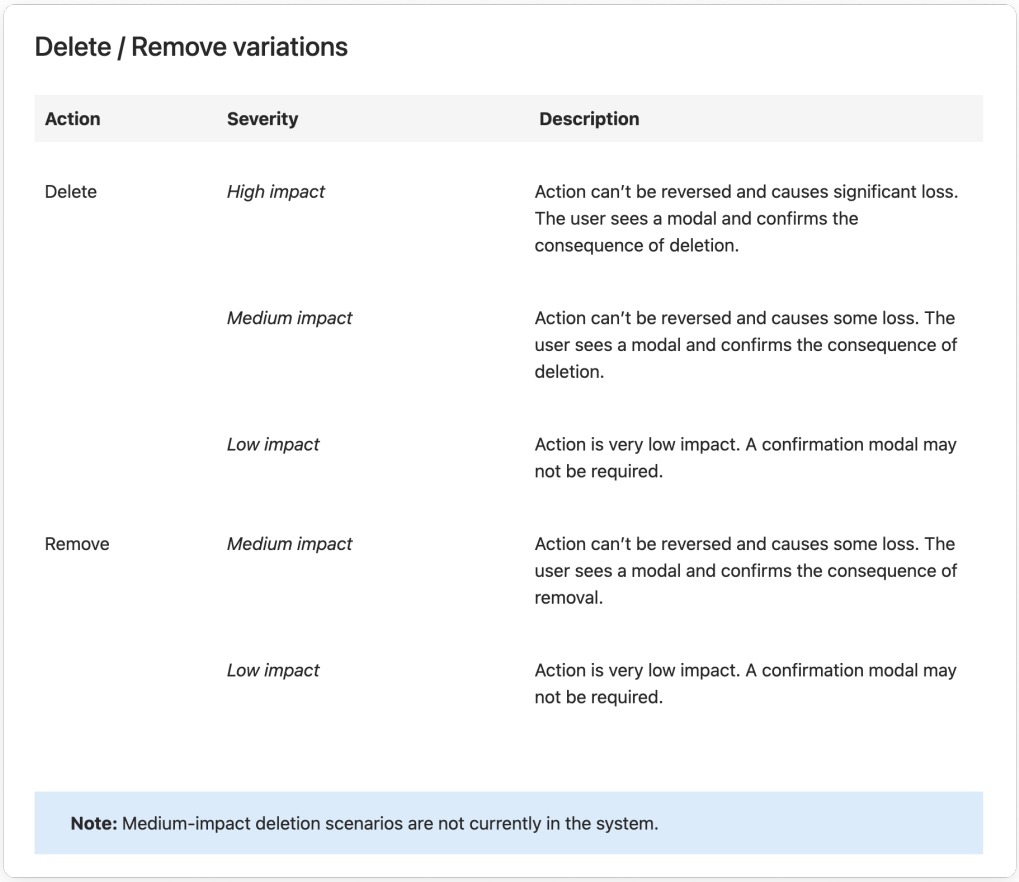
Our design application will store the visual design source of truth for things like component construction and token adherence. This is the place where designers are creating their user interface compositions and the library of usable components is defined and held. As we discussed before this is the same location where we apply our tokens to our components, allowing developers to properly transfer them to code. We can see in this example a few key pieces of information that have been included in both the Agnostic Design System and Product Kit design library. First there is a badge at the top of the documentation card that shows the status of that component. This is a quick way for stakeholders to tell whether a component is a work in progress, being coded, or is ready for use. Next we have links to various documentation sources. These are shortcuts that help connect all the documentation locations to one another. In the ADS you may only need links to the code since the ADS is not supposed to have any user experience usage documentation. Where the Product Kit has a link for those docs and the code for its specific implementation. There is also a short description of the component, what it is, and any specific prescriptions around its usage to give a high level overview. Finally we have the component that the designers end up using in their designs and developers reference when developing the application.

The second set of documentation can be put in a location like Confluence, Zeroheight, or Notion. This documentation is only needed for the Product Kit side of things as the Agnostic Design System has no user experience guidelines to document, those are defined by the product implementing it. When searching for a documentation site consider looking for features that let you include code previews or design application embeds. Being able to connect your documentation to other sources of truth allow for teams to spend less time on maintenance since the documentation will update to match the content you have embedded. This documentation set should include three primary groups of information.



The first group revolves around components. Document all the available components along with their usage, interaction, and accessibility details. What variants are available? What does the component do? Are there things that should be avoided with the component? If your

documentation site supports the inclusion of design files or code then add those so viewers are able to see the documentation in context and get a better feel for how things work.

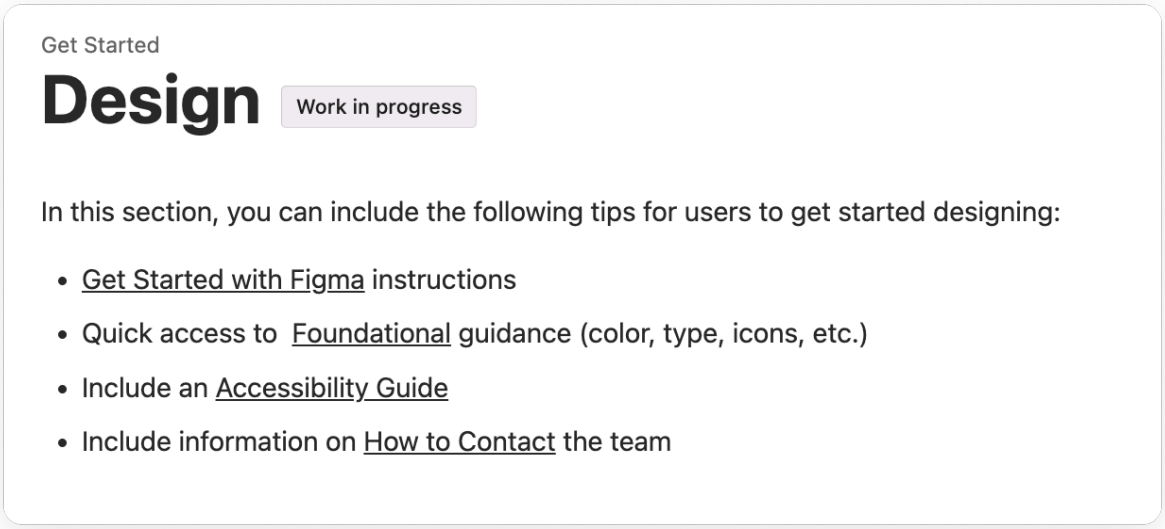


Delete / Remove variations

Action	Severity	Description
Delete	*High impact*	Action can't be reversed and causes significant loss. The user sees a modal and confirms the consequence of deletion.
	Medium impact	Action can't be reversed and causes some loss. The user sees a modal and confirms the consequence of deletion.
	Low impact	Action is very low impact. A confirmation modal may not be required.
Remove	*Medium impact*	Action can't be reversed and causes some loss. The user sees a modal and confirms the consequence of removal.
	Low impact	Action is very low impact. A confirmation modal may not be required.

Note: Medium-impact deletion scenarios are not currently in the system.

Patterns and standards make up the next group. This group consists of all the product experiences we want to make uniform and repeatable. Ensuring your application is predictable is important to reducing friction. If users are following 5 different processes to find a page or delete an item they will never be able to feel comfortable in the application. This unification starts and ends with the designers crafting the experiences so we want to make sure they are all following the same standards. Here we document experience flows like what errors to show when, how menus work, or which type of button to use when. In the case of the above example we are surfacing the details around how user deletion of information from the application works. Each severity level and action type is neatly defined so each stakeholder can fully understand each scenario and apply it to their situation accordingly.



The last set of documentation revolves around processes and procedures. The purpose of this is to inform individuals who are new to a system or contributing to it. It acts as an introduction to the Product Kit and outlines its purpose and whether or not it utilizes other systems like an ADS. Processes that could be included are how designers get new components into the system, how to add new icons or assets, or what to do when there are missing styles. Consider the questions you often get around the design system as a good place for sourcing what sort of process documentation should be included here.

Documentation serves the important purpose of connecting all of the stakeholders together with unified language and goals. While each stakeholder may have their own internal documentation, the way designers have their design application for the minutiae of component construction, these also serve to inform the set of collaborative documentation that keeps everyone on the same page. As each group has questions they will be able to go to the documentation to find the answers. Instead of playing telephone with individuals trying to find the answer to your question, this standard set of documentation will have everything you need.

Chapter 6: UDS Benefits

We’ve covered each of the major pieces of a Universal Design System. Let’s bring them all together and discuss the short and long term benefits of this system methodology.

As discussed, the Universal Design System seeks to solve one of the most complicated problems in product development, information management. With developers, designers, and product individuals all having a hand in the creation of digital experiences keeping each of them on the same page is paramount to smooth and efficient operation. We introduce an Agnostic Design System to handle the rapid changes in the digital world and reduce the overhead required when recreating the same component for multiple products. The Product Kits sit on top of the agnostic system and utilize these benefits to operate at the speed of their own timelines and under one unified design language. While the documentation for each brings everything together for all to collaborate and contribute to the system as a whole. All throughout we have embedded flexibility so we are prepared for whatever the future has in store. Along with these there are other benefits that come from each part of the UDS on their own as well as together.

Product Decoupling

When we have multiple products, or digital applications, being worked on we encounter issues around priority collisions. One product needs a dashboard component by next month, another needs a header in a week, and the other needs a new button. In a normal system each request would be considered based on a set of criteria to determine who got their work done first. In the end one or more groups is losing out as timelines are often hard to adjust. The UDS decouples the products from their code in order to allow each to work at pace with their own timelines. If the component exists in the ADS then that product can go ahead, theme it, then use it in their app. If the ADS does not have the component then the products don’t need to wait for the ADS to catch up. They are given the autonomy to generate their own components with or without the ADS components. Since the Agnostic Design System contains many low level atom components like buttons, inputs, and controls, products are able to utilize those to stay connected with the agnostic foundation but produce their own product specific solution. Once the ADS has free time those product specific components can be considered for incorporation into the ADS for use by other products in the future. As well, the Agnostic Design System is based on universal development standards which allows for any development framework to be used. Removing the roadblock that comes with compatibility issues between frameworks chosen by products. This decoupling allows everyone to work asynchronously but never lose out on the work done by one group.

Communication Streamlining

The Universal Design System presents a shared language based on foundational product development concepts. As the ADS is not subjected to a specific framework but rather fundamental web standards, the ability for others to understand the underlying parts is greatly improved. Stakeholders don't need to understand how React works versus how Vue works. The most anyone needs to know is that the ADS offers a set of components that have the ability to look however you want. The purpose and architecture of the Agnostic Design System is so simple that it becomes much easier for each stakeholder to fully understand its purpose. What you see is what you get. Where the complexity starts to come in is with the Product Kits and their set of documentation and product specific assets. Although complex, the organization of product documentation should be thoughtful and intentionally executed. As we've discussed there are group based information sources that serve to provide fine-grain detail for each stakeholder. But there is also the shared documentation source that brings all of this disparate information together for all to benefit from. The UDS distills documentation down to prevent over definition, loose ends, and information entropy. The more you interconnect documentation through the use of embedded design and code files the more accurate the system becomes. Each stakeholder is able to glean the same information in order to make informed decisions.

Overhead Reduction

The ADS has the responsibility of doing a lot of the leg work up front and free from distraction. With core components pre-defined the effort required is greatly reduced as products now have a Lego kit of sorts to work from instead of spending all that time re-developing the same button over and over again. The agnostic system takes the work that would be multiplied per product and does it once. This has the added benefit of opening the door for technologies to be incorporated universally rather than per product. As new business requirements come out that all products have to subscribe to going forward, the ADS can implement it within the foundation of all the products. Doing the work once instead of multiple times. Where products would usually be left to their own devices to do the bulk of the implementation, the ADS's sharing capabilities reduces this overhead.

Accessibility From The Start

A good example of overhead reduction can also be found in the inclusion of accessibility within products. Good practice when creating digital experience is to always include accessibility

features from the get go. But unfortunately this is not always the case. Whether it be timeline restrictions or a “lack of need” many products ignore these requirements. Since accessibility features can be supplied at the foundation of every product via the ADS, the excuses for its lack of inclusion start to wane. The need for every single product team to understand it is cut down as well since the ADS is essentially a templated system, meaning everything in it is provided to everyone using it. It doesn't provide full coverage for standards like WCAG, as there are contextual requirements around accessibility, but it massively reduces the friction to becoming accessible.

Rapid Iteration

With a foundational framework like the ADS, you are now afforded the ability to venture into new territories previously deemed too costly. Having all this overhead pre-done the cost to spin up new product ideas is vastly reduced. New and experimental apps can be spun up to test out ideas in a more tangible format instead of solely relying on design prototyping. This also extends into the creation of tooling specific to the productivity of the business. Applications can be created for all parts of the business instead of needing to invest in costly one-off solutions or third-party integrations that lock you into their roadmap and technologies. The Agnostic Design System serves as a powerful prototyping tool and cost analysis driver as you can more easily determine the work required by actually testing out its implementation. Allowing for easier determination of viability and solution sources.

No matter the direction you take your design system, consider including the aforementioned concepts in your solution. Each is important to a cohesive and flexible design system that will stand the test of time. They allow for priorities and concerns to be properly assessed and acted upon in a structured way.

Chapter 7: Conclusion

The design systems of today have served their purpose, but as products increase in scope and complexity it is clear the way forward is found with the Universal Design System.

We rely on universally understood standardization in our everyday lives to maintain a sense of consistency and predictability in our experiences and decisions. When we are hit with variability our senses are heightened and a sense of uncomfortability sets in as we use previous knowledge to try and ascertain what the outcome will be. While we can't control everything, like in the case of intersections where the movement of every car is the choice of the driver, we can set standards that every driver can understand and use to infer these outcomes with greater precision. Just like the Department of Transportation's standardization and implementation of traffic control infrastructure. Drivers are afforded with the confidence that stop signage, like stop signs or traffic lights, will appear the same and have the same meaning everywhere they go. This requirement for user confidence extends to your product in order to maintain a smooth and enjoyable experience. Products without predictability instill distrust in its users, ultimately leading to user dissatisfaction. In order to achieve this predictability we have to be able to corral all the parts of the product in order to standardize it and keep it standardized.

The Universal Design System steps up as the solution to this problem. We cover all the primary points of a standardized system by utilizing a foundational layer of components and functionality known as the Agnostic Design System, and the application specific implementation through the use of Product Kits. The principles of each are useful with or without one or the other. But combined they form an impenetrable system capable of handling any situation thrown at it. The UDS takes the common concepts of today's systems and sets them up to work with one another, playing off of one another's strengths.

As we've discussed much of this system at a higher level, my hope is that you will be able to take these concepts and implement them into your own system. Whether that be individual application of concepts or a full rewrite of your existing system, the benefits of a UDS cannot be understated. For a deeper dive into the implementation of these concepts consider checking out my other books. There I cover topics like design token structures, theming, design library component construction, designer experience patterns, and more.

Terminology

Design Systems

An amalgamation of design, development, and business requirements that come together to form a repeatable set of assets to build digital and physical products. The most well known parts of a design system are the use of standardized colors, typography, spacing, and components.

Agnostic System

Many systems out there stick themselves to a particular framework. Whether it be React, Vue, Angular, or one of the many other options. A system that is agnostic means that it does not need to adhere to any one framework. It is set up in such a way that its core is not reliant on a framework and can be morphed to fit whatever framework is desired, if any are.

Patterns

A type of documentation that outlines the preferred usage of a component or operation of a feature experience in a specific product. These help keep all similar scenarios in a product operating in a predictable manner. Making designing easier and the UX of a product more consistent.

Accessibility

This is the practice of making your product usable for individuals with disabilities ranging from mental to physical.

Components

These live in both the design and the development realm but should be congruent. They are your buttons, checkboxes, cards, etc. These arise from the combination of your foundational design system principles of color, spacing, and type and the functionality provided through code.

Tokens

Often referred to as “design tokens”, these are variables that are set to various styles in the system. You can introduce logic and various parent/child relationships to enable theming and other advanced styling abilities.

Master Component

These are the UI components that act as templates and dictate how any component variant using it looks.

Component Variant

This means two things. First, the master component does not have to be just one component nor should it be. Many components require states such as hover and disabled to be defined visually. The variants allow you to do so by assigning states to different types of the same master component. The other purpose of a variant is to allow designers to use the component in their designs but remain connected to the source of truth and get any updates that the master component gets from its maintainers.

Component Parts

Parts are often not used by development or the end designers but serve as construction helpers when putting together components in the design software. These are pieces that make up the master or variant components but are not usually intended as a usable component alone.

Blueprints

Variations on core components to produce solutions for specific use cases. For example you might have a menu component but want specific menu setups to be available to developers and designers. Everyone that is going to be using a settings menu you might want them all pulling in the same exact menu. Blueprints plainly lay out how this should look so both groups are not starting from scratch. Much of this is determined by product patterns.

“Replace Me”

A construction helper, similar to slots and are primarily for designers. These are used in places like cards, tiles, or messages. Places where a large area of blank space exists where any number of compositions can be placed within. They also serve to enhance the design experience and allow for more accurate and realistic designs to be produced.

Slots

These are predestined locations inside a component that other assets or components can be placed into. They allow for customization by designers so they are able to produce solutions specific to their situation and act as a construction aid in the design process.