**-Chapter 1: The Mise en Place Philosophy(1-11)**

Mise en Place; **mutfakta her şeyin yerli yerinde olmasına verilen Fransızca kökenli bir kelime**. Yani yemek yapmaya başlamadan önce yapılan ön hazırlığın tümünü kapsayan bir terim. **Wikipedia'ya göre mise en place, "her şey yerli yerinde" anlamına gelir.**

* **Software Is Different, Right?**

Yazılım geliştirmenin bu göreceli “yeniliği”, özellikle front-end geliştirmede hem özgürleştirici hem de sorunlu olabilir! Tekrar tekrar, eğlenceli ve heyecan verici bir şekilde başlayan bir proje, yıllar süren geliştirmelerin ardından bir kabusa dönüşüyor. Birkaç yıl sonra kendimizi aynı durumda bulmak için genellikle uygulamalarımızı yeniden yazıyoruz. Bu kitap boyunca ilerlerken, yazılımımızı hem eğlenceli hem de sürdürülebilir kılmak için kullanabileceğimiz bazı stratejileri tartışacağız.

* **Good Software**

*We build our computers [systems] the way we build our*

*cities—over time, without a plan, on top of ruins.*

—Ellen Ullman, programmer and author

Hiç bir kod tabanına serpiştirilmiş aynı işlevselliğe sahip kod buldunuz mu?

Uygulamanın tamamen ayrı bir bölümünü bozduğunuzu öğrenmek için bir uygulamanın bir bölümü için stilleri hiç güncellediniz mi?

Hiç bir kod tabanını sürdürmenin zorluklarını rewrite gerektirecek kadar ezici bulan bir takımda bulundunuz mu? (Bonus soru: rewrite, karşılaşılan tüm sorunları çözdü mü?)

The sad thing about these responses is that things don’t need to be this way.( Bu yanıtlarla ilgili üzücü olan şey, işlerin bu şekilde olması gerekmemesidir.)

Yazılım geliştiriciler olarak bizim ilham alabileceğimiz birçok sektör var, ancak özellikle sevdiğim bir tanesi mutfak dünyası —mise en place.

* **What Does This Have to Do with Software? (Bunun Yazılımla Ne ilgisi Var?)**

Adam Savage’ın mise en palce Philosophy hayranıdır.

*For all the alchemy that goes into building something, the*

*magic of making is only possible because of the many repetitive*

*processes we endure in preparation for final assembly.*

—Adam Savage

Başka bir deyişle, harika bir sonuç ancak bir şeyler inşa etmek için attığımız küçük adımlar sayesinde mümkündür.

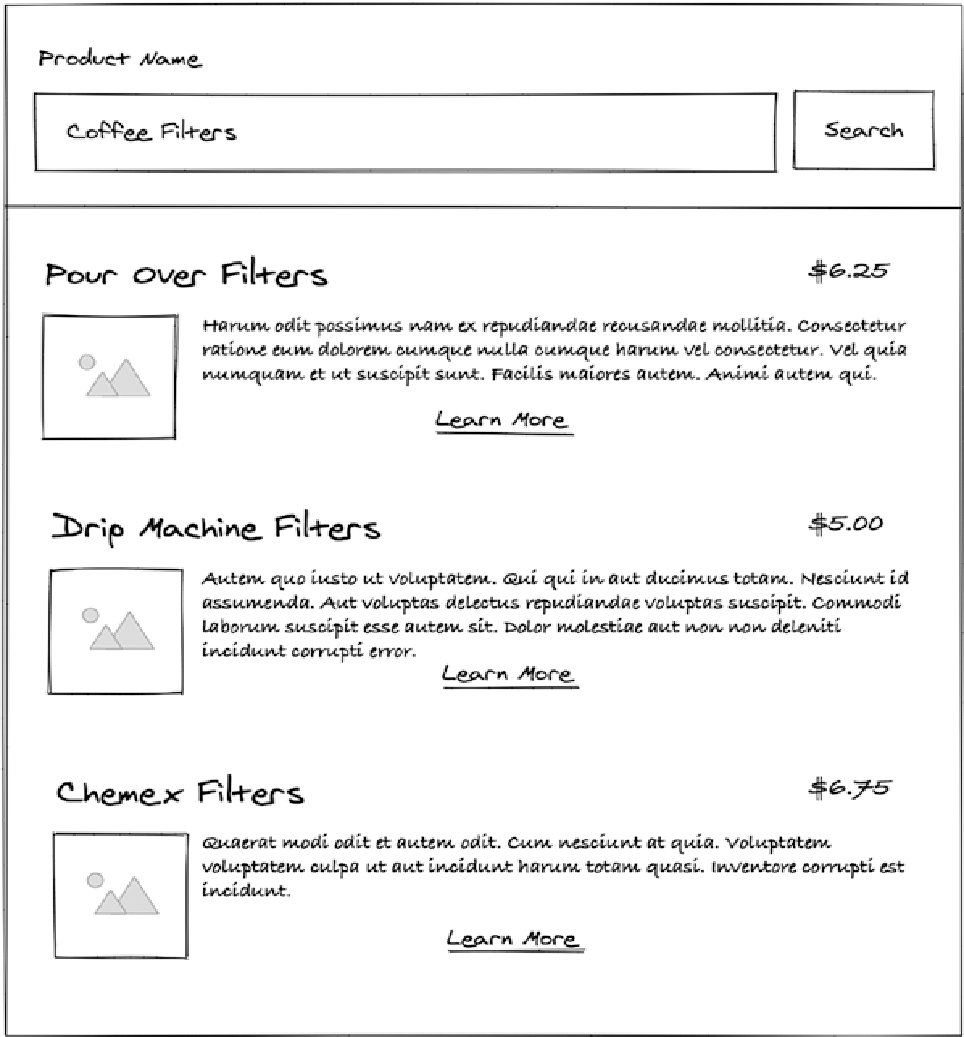
Yazılım geliştirmede, üstlendiğimiz birçok **repetitive processes ( tekrarlayan süreç)** vardır. Bu süreçler her zaman harika bir sonuca hazırlanmamıza yardımcı oluyor gibi görünmeyebilir, ancak ayrıntılara dikkat ederek ve bir **uygulamanın (focus on the individual aspects of an application ) bireysel yönlerine odaklanarak harika** sonuçlar elde edebiliriz. Uygulamamızın bazı yönlerine **bireysel olarak odaklanabilmemizin** bir yolu, **components** etkin bir şekilde kullanmaktır.

* **Components**

Components are a central concept to Angular, Vue, React, Svelte, and others. Most browsers even support native web components!

Components, uygulamalarımızı bir dizi daha küçük uygulama gibi görünen parçalara ayırmamıza izin verir. React gibi framework’ler, geliştiricilerin bileşenleri diğer bileşenlere dönüştürmesine olanak tanıyan bir **interface** veya **contract** sağlar. Bir dizi bileşenden sayfalar veya ekranlar oluşturabiliriz. Şekil 1-1'de gösterilen bir ürün arama modeline bir göz atalım.

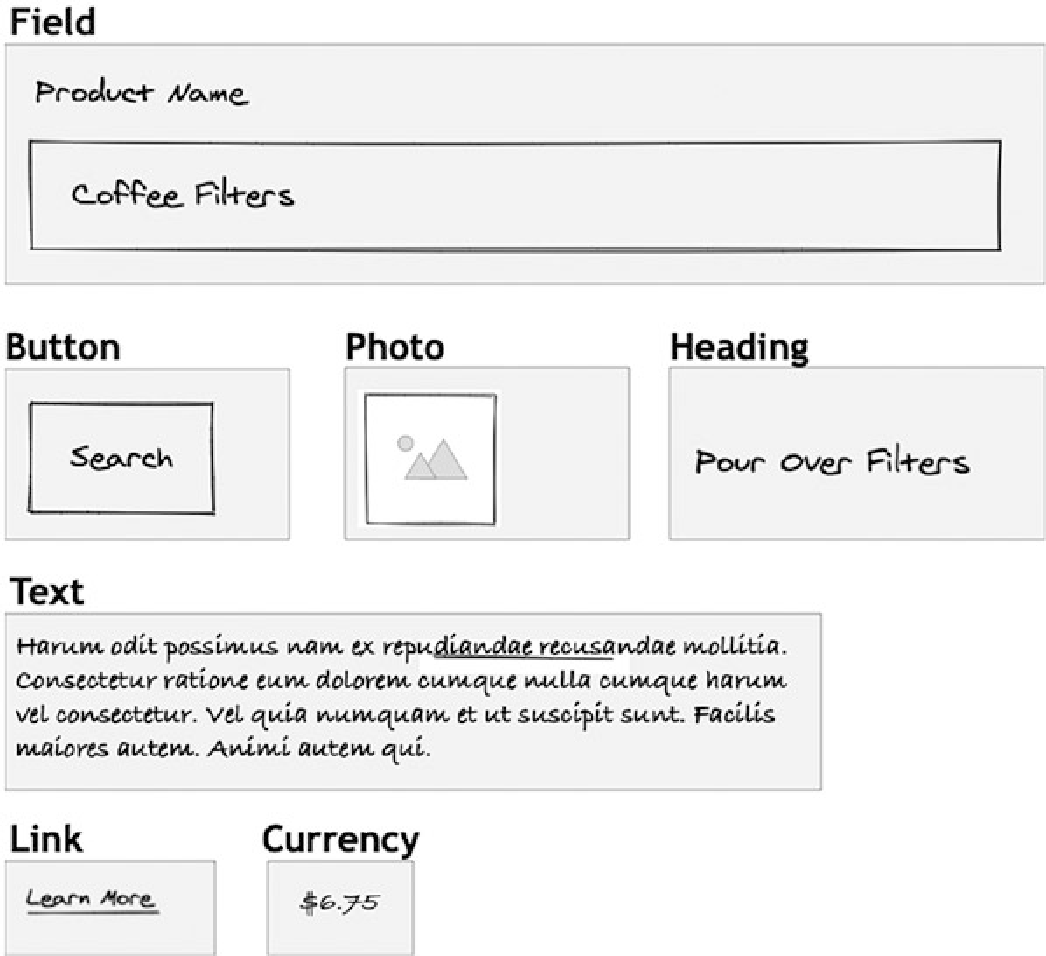
Bir dizi component sayfalar veya ekranlar oluşturabiliriz. Şekil 1-1'de gösterilen **a product search mockup (bir ürün arama modeline)** bir göz atalım.



Bu ürün aramasını nasıl yapılandıracağımızı düşünüyorsunuz? Bu maketi **tek bir sayfa(single page)** olarak mı görselleştiriyorsunuz, yoksa bir dizi components olarak mı görüyorsunuz?

Burada yanlış bir cevap yok, ancak çoğu durumda **bunu product search input** ve **search results**. ayırabiliriz.

Continuing on this path, bu components altda gösterildiği **gibi field (input ve input label), button, Photo, Heading, Text ekranına ve Link** dönüştürebiliriz.



Pişirmedeki malzemeler başka yemeklerde kullanılabildiğinden, bu components diğer ekranların bir parçası olarak kullanılabilir.

Günümüzün çoğu için temel olan bu bileşen kavramı frontend frameworks için harika. Ne yazık ki, **uygulamaları oluşturmak için bileşenleri kullanmak ve bileşenleri etkin bir şekilde kullanmak aynı şey değildir.**

* **Meetings Don’t Ensure Success (Toplantılar Başarıyı Sağlamıyor)**

Büyük bir zorluk, ekiplerin uyumlu olmasını sağlamak ve aynı görünen benzer bileşenleri kullanmaktı. Birkaç haftada bir, bu ekiplerden bazı geliştiriciler, ekiplerinin karşılaştığı zorlukları ve başarıları tartışmak için bir araya gelirdi. Hemen hemen her toplantıda birileri "Bu sorunu çözen harika bir bileşen yaptık" derdi. Bir başkası, “Ekibimiz bunlardan birini de yaptı” derdi. **Bazı durumlarda, aynı amaca hizmet eden dört componets örneğine sahip olduğumuz ortaya çıktı!**

Bu toplantı, ortak, gelecekteki hedeflere uyum sağlamak için harikaydı, ancak bu gerekli tutarlılığın sağlanmasına yardımcı olmadı. Yalnızca bir dizi componentden uygulamalar yapmanın değil, aynı zamanda **hangi bileşenlerin var olduğunu iletmenin daha iyi bir yolu olmalıydı. (storybookdan bahsediyor galiba )**

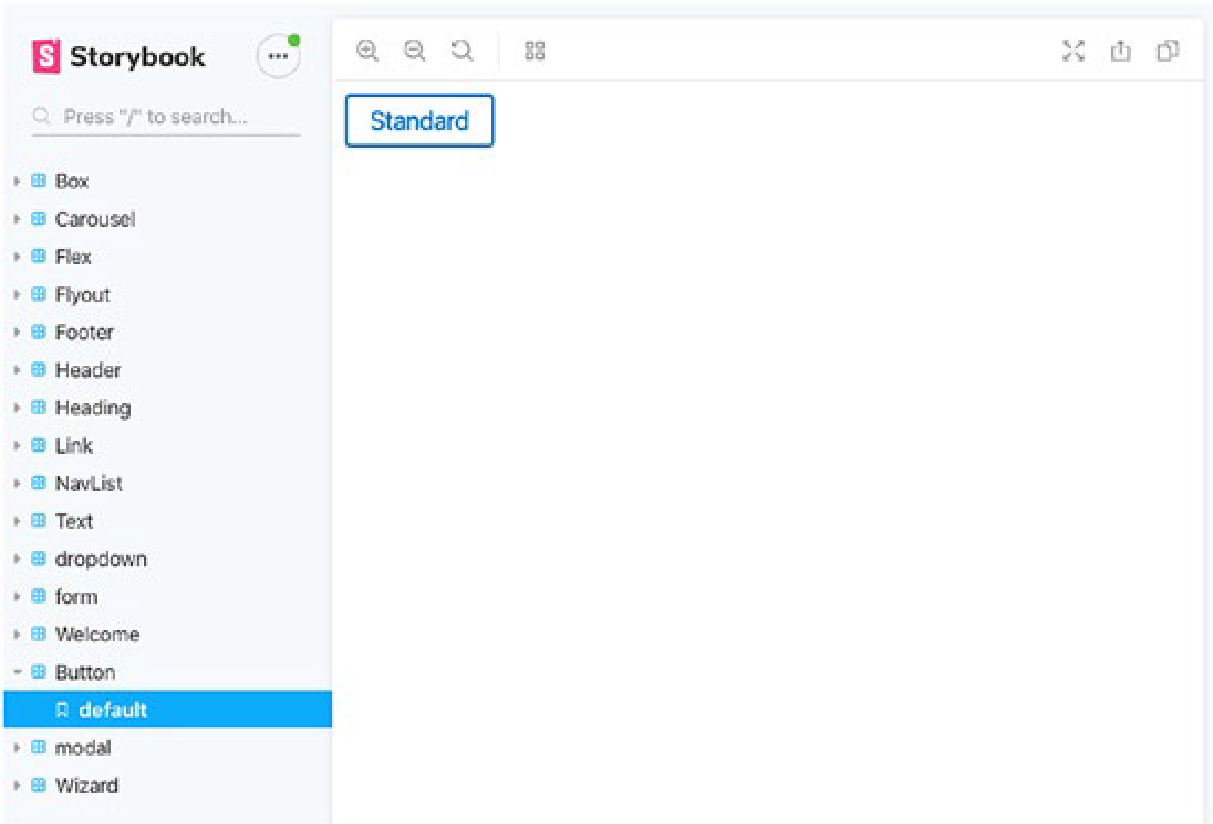
* **The Workspace (Çalışma Alanı)**

Component’leri ayrı ayrı oluşturmanın sayısız faydası vardır, ancak akılda tutulması gereken birkaç önemli nokta vardır:

1. Bir **component workspace**(çalışma alanı)kullanmak, gerçekten **reusable(yeniden kullanılabilir)** componentler yazmamıza yardımcı olabilir. Bir sayfanın parçası olarak bir component üzerinde çalışırken tunnel vision sahip olmak kolaydır. **Sayfanın** **responsibilities (sorumluluklarını)** düşünmek yerine, **component sorumluluklarına** odaklanabiliriz.
2. Her seferinde bir component’e odaklanmak, inşa ettiğimiz component’in kalitesinden emin olmamıza yardımcı olur. Bir şefin mutfakta mükemmel bir sonuç elde etmek için bir yemeğin tek tek parçalarını test etmesi gibi, yüksek kaliteli component’ler daha iyi uygulamalara yol açar.
3. Bir component workspace diğer geliştiriciler, tasarımcılar ve paydaşlarla iletişim kurabilir, öğeler bir sistemde mevcut olduğunda. Bu iletişim, componentlerin yalnızca başkaları var olduklarının farkında olmadığı için **duplicated (çoğaltıldığı)** senaryolardan kaçınmamıza yardımcı olur.

* **Storybook**

Bu kitapta we will use Storybook as our component workspace. Storybook, building component libraries yardımcı olan açık kaynaklı bir projedir.



Storybook'u kullanarak workspace’le **stories** aracılığıyla etkileşim kurarız. **Stories**, belirli bir durumdaki bir component’in görsel temsilleridir.

By default, we see our stories in a treelist view with all available component stories on the left and the workspace on the right. Uygulamamızı Storybook'u kendi başına başlatmak için bir komut dosyasıyla(script) çalışacak şekilde yapılandıracağız. Bu, npm run storybook veya yarn storybook gibi görünebilir.

* **Key Takeaways (Önemli Bilgiler, Önemli Çıkarımlar)**

-Bu bölümde, bugün frontend karşılaştığımız bazı zorluklardan bahsettik. **Consistency(Tutarlılık),** **communication(iletişim)** ve **developer productivity (geliştirici üretkenliği,** **verimliliği)**, büyük bir codebase’de elde etmek zor olabilir.

- Diğer endüstrilerin, tutarlılık ve kaliteyi verimli bir şekilde elde etmeye yardımcı olacak organizasyon ve sistemler için stratejileri vardır.

- Mise en place, yazılım geliştirmede kullanabileceğimiz mutfak dünyasında kullanılan böyle bir stratejidir. **Mise en place, bir yemek hazırlamadan önce devam eden organizasyon ve hazırlığı ifade eder.**

**-** Bir **workspace**, ticari mutfaklarda mise en place'in ayrılmaz bir parçasıdır. “Mise en place”i, kod tabanımıza uyguladığımız hem **mimari(architecture)** hem de **organizasyon(organization)** olarak düşünebiliriz. **Geliştiriciler(Developers)**, ayrı ayrı codebase bölümlerine odaklanmaya yardımcı olmak için bir çalışma alanından yararlanabilir.

- Storybook, kodumuz için bir **çalışma alanı(workspace)** olarak hizmet veren mükemmel bir **araçtır(tool).**

**-Chapter 2: Configuring Our Workspace(12-27)**

Taklit etmek istediğimiz, benzemeye çalıştığımız (emulate) the mise ne place felsefesini biraz daha anladığımıza göre, şimdi koda geçelim. Bu bölümde, we’re going to ensure(sağlamak) our computer is set up(kurmak) and configured(yapılandırmak, ayarlamak, konfigür etmek) for our success.

* Sistem Gereksinimleri

-Node.js sürüm 8 veya üzeri

-JavaScript dosyalarıyla çalışmak için kendinizi rahat hissedeceğiniz bir düzenleyici(VSCode)

------------------------------------------

**Not**: **Production applications** için Node.js'nin yalnızca **çift sayılı(even-numbered) ana sürümlerini kullanmayı** düşünmelisiniz. Node.js belgelerine göre, **çift sayılı ana sürümlerin (LTS**  **Long Time Support)uzun vadeli desteği vardır** veya **30 ay boyunca hata düzeltmeleri alırlar.** **Tek sayılı(Odd numbered**) sürümlerin yalnızca **altı aylık** desteği vardır (<https://nodejs.org/en/about/releases/>).

------------------------------------------

Note.js direk oarak kullanmıcaz. Tools ve librariesler kullanacaz. CLI tool for managing installed Node packages) and Create React App (CRA), depend on Node.js.

* **Installing Node.js**

Bazı codebase’ler belirli node.js versiyonlarına bağlı oluyor farklı node versionlar kullanıncada sorun çıkıyor. İşte bu sorunlardan kurtulak için node version manager’ları(nvm) kullanacaz.

* **Version Managers**

Version manager; bilgisayarımıza birçok Node.js kurulumunu kurmamızı ve yönetmemizi sağlayan bir yazılımdır.

Note: Node Version Manager (NVM) and Node Version Manager for Windows are similar

İlerde windows ve mac için node v10.20.1 kurulumlarını gösteriyor.

* **Leveraging Node.js Built-In Utilities**

Node.js de built-in bulunan 2 tane tool’dan bahsetti**.( Npx ve npm )**

**npx tool,** cihazımızda npm kayıt defterinde depolanan herhangi bir command-line library çalıştırmamıza izin verir. Let’s take a look at how we can create a React app leveraging npx:

# with npx we can create a react app quickly

> npx create-react-app MyApp

Veya

# previously a command line tool would need to be installed before it could be used

> npm install -g create-react-app

. . .

> create-react-app MyApp

* **Creating Our React Application**

->cd rocket-coffee

->npm run start

* **Adding Our Workspace**

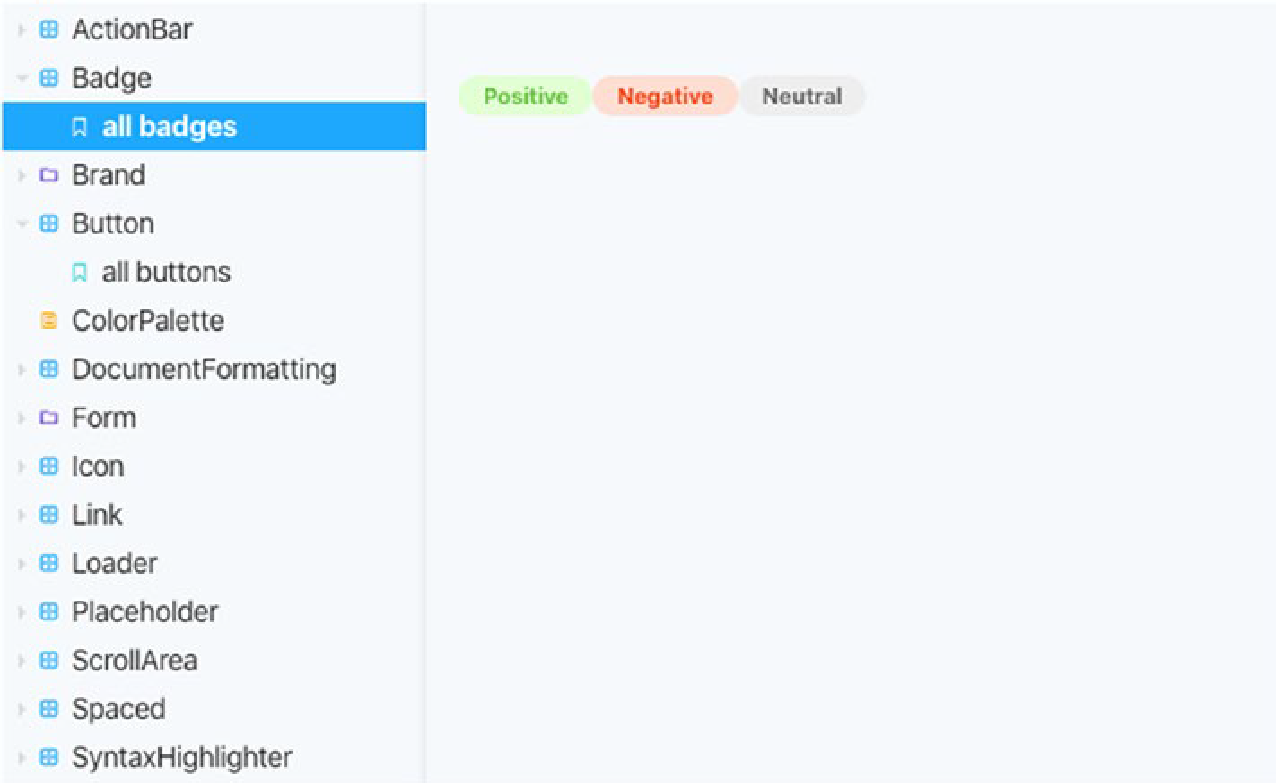
This component workspace should be a little bit like the different

stations in a restaurant where culinary artists build the elements of a dish.

Similarly, our **component workspace** will be where we focus on **individual elements of a page/screen.**

* **Storybook**

Storybook, building component libraries için mükemmel bir araçtır. Bu tool, story concept etrafında düzenlenmiştir. Bir story, bir component’in benzersiz bir durumunu temsil eder.( A story represents a unique state of a component.) Örnek bir storybook hızlı bir bakış allta görülmektedir.



Resim’in sağ tarafında **story’nin görsel** çıktısını görüyoruz. Her bileşenin birçok story’si olabilir, ancak bu örnekte **yalnızca bir tane story’**imiz var. Sol tarafta, ağaç **görünümünde(tree view**) düzenlenmiş tüm olası storylerin bir listesini görüyoruz. Bu ağaç görünümünde farklı bir öğe seçerek aktif hikayeyi değiştirebiliriz.

* **Installing Storybook**
* npx -p @storybook/cli sb init
* npm run storybook ile çalıştırılır
* **Key Takeaways (Chapter 2 için Önemli Bilgiler)**

Bu bölüm tamamen **hazırlıkla(preparation**) ilgiliydi. Çoğu durumda, Node.js'nin sistem genelinde bir sürümünü kullanmak, kaçınmak istediğimiz sorunlara neden olabilir. Node.**js'yi, bizi framework’ün** belirli bir sürümüne bağlamayan bir **sürüm yöneticisi(version manager**) aracılığıyla kurduk.

Create React App aracılığıyla bir React uygulaması oluşturduk ve **varsayılan Storybook component çalışma alanını(workspace)** başlattık. codebase hazırlamak, uygulamalarımıza güç veren kodu yazmak kadar eğlenceli olmasa da, harika bir sonuç elde etmek için ayrılmaz bir adımdır.

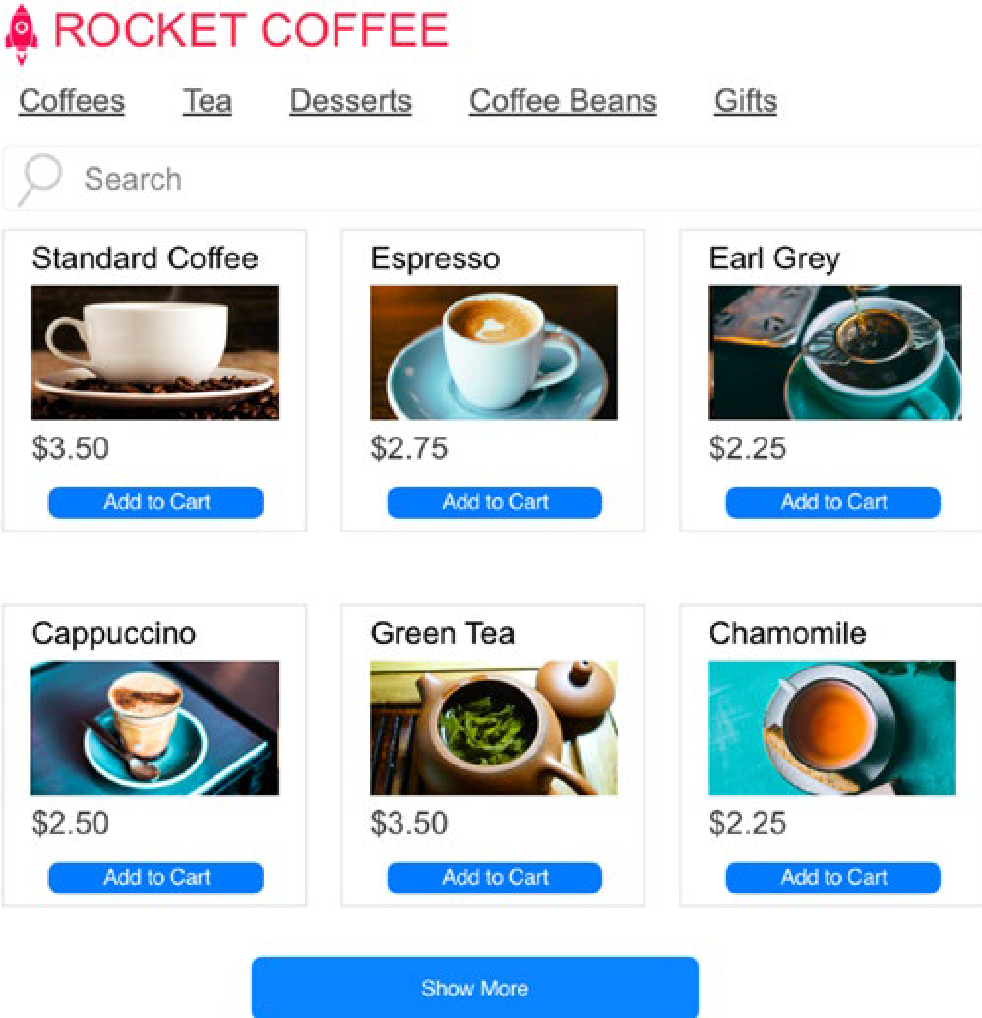
Her şeyden önce, hazırlık başarının anahtarıdır.

(Before anything else, preparation is the key to success.)

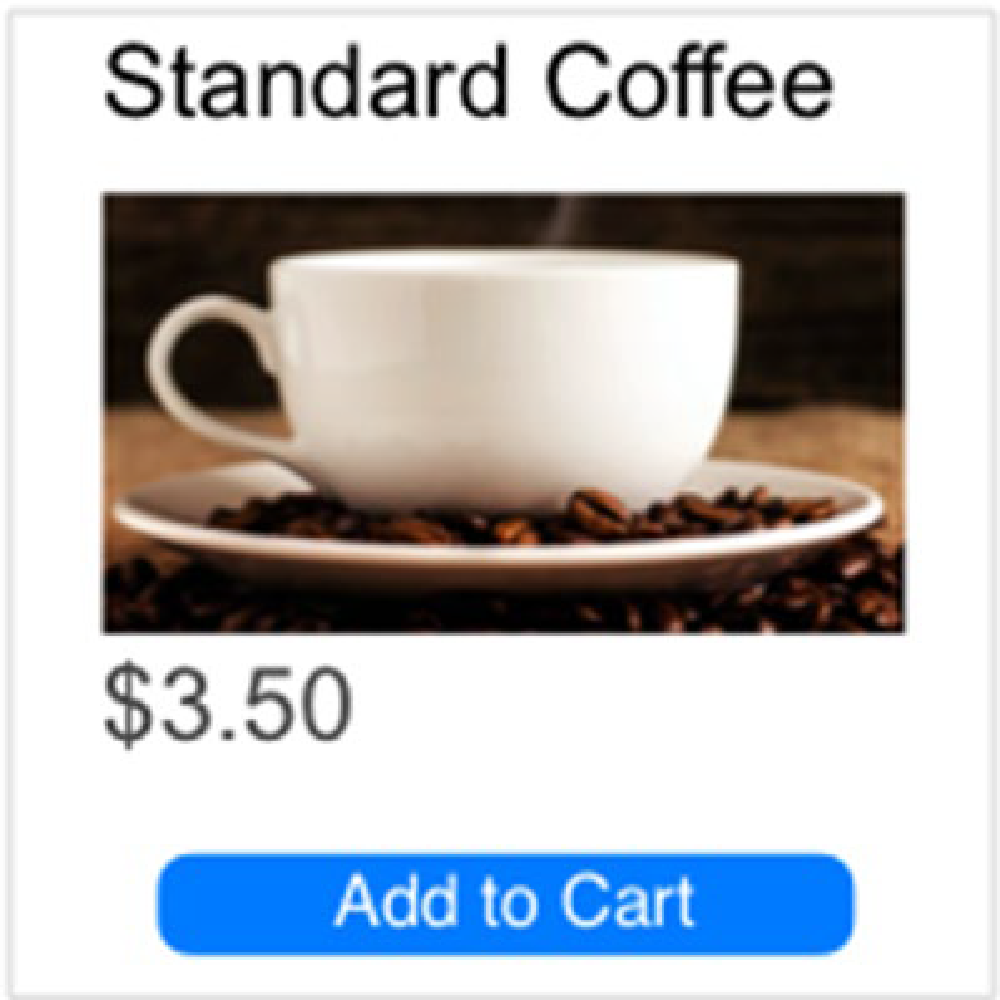
—Alexander Graham Bell

**-Chapter 3 Our First Storybook Stories(29-45)**

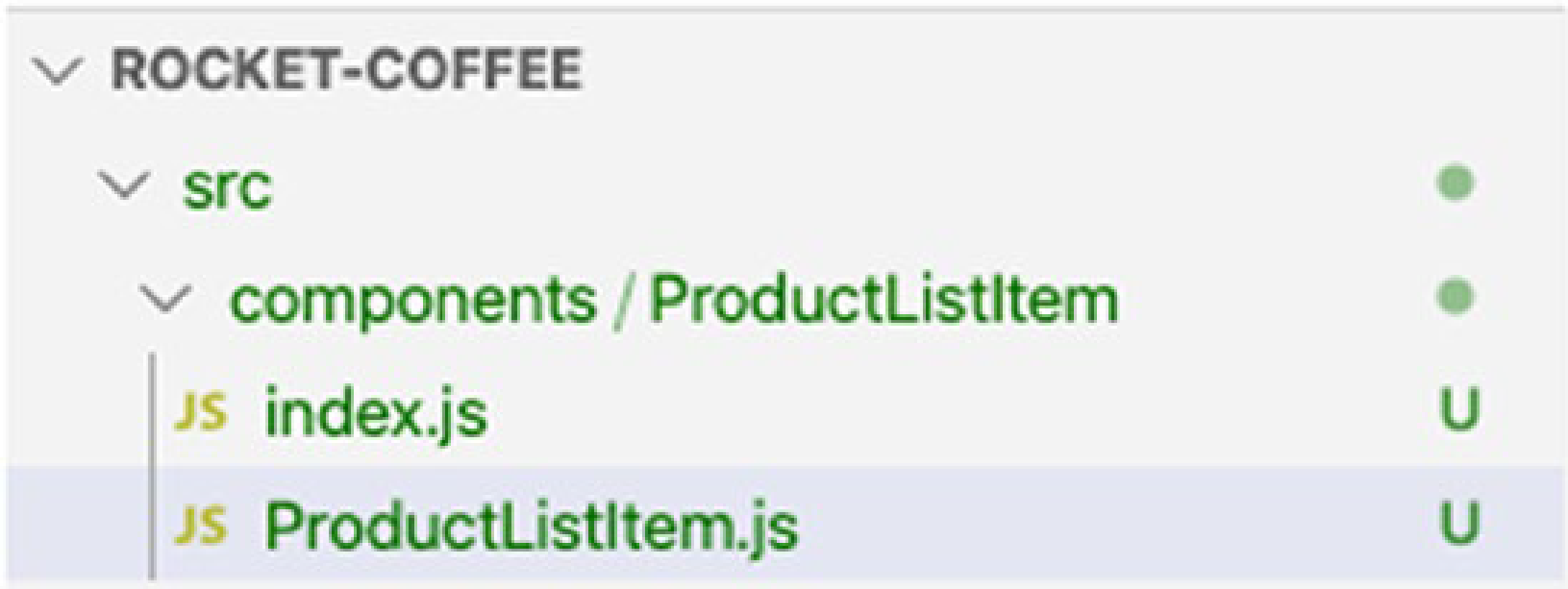
Rocket Coffee adında bir kahve/çay dükkanı için bir e-ticaret mağazası oluşturacağız. Tasarımcımızın bize üzerinde çalışabileceğimiz bazı tasarımlar verdiğini, ancak bunun dışında pek bir yön vermediğini farz edeceğiz. Şekil 3-1'deki ürün listesine bir göz atalım.



We don’t want to **tackle(ele almak**) this entire page at once. For now, we’re going to focus on **building a component for a single product item**.



*Product item*

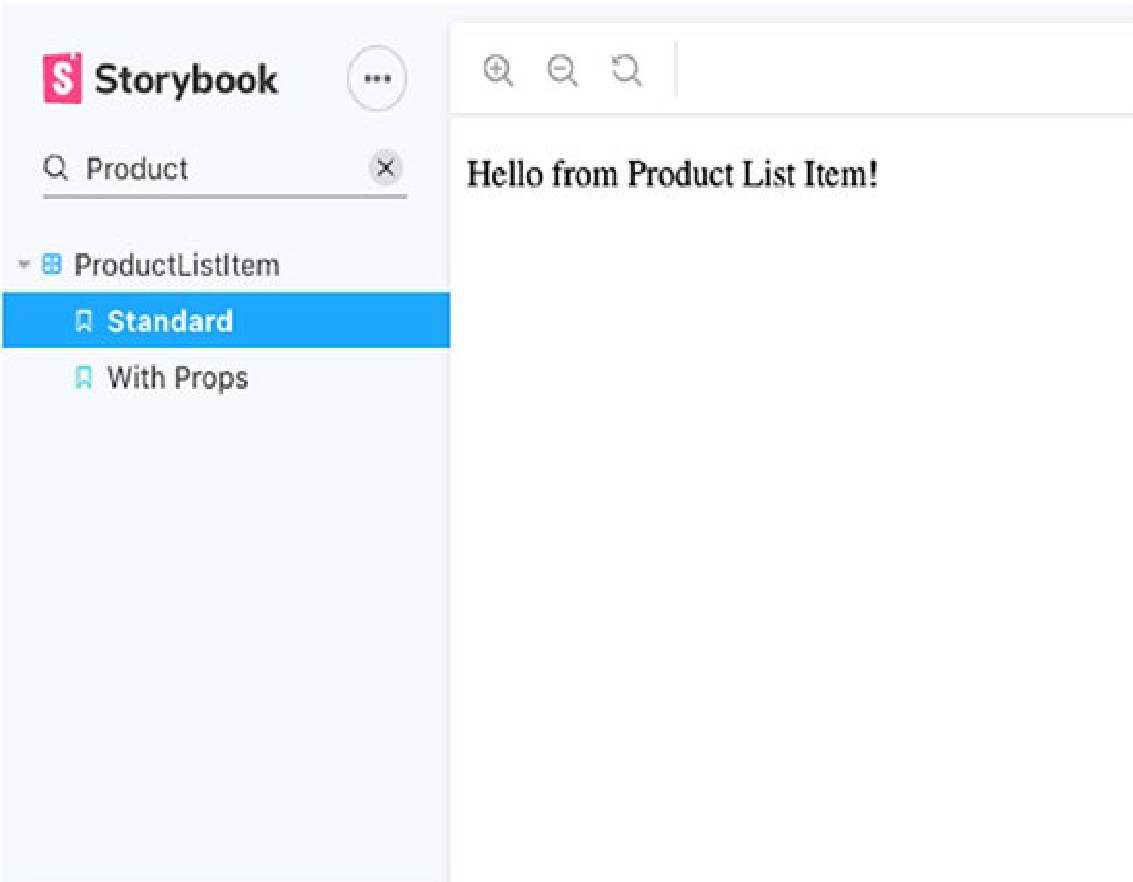


Not: Tüm kodumuzu index.js dosyasının içine yerleştirebilir ve temelde doğrudan geçişli bir dosya oluşturmaktan kaçınabiliriz. Kodumu çok özel bir dosya adıyla saklamanın, bu kodu daha sonra bulmama yardımcı olduğunu sık sık fark ettim. İyi yapılandırılmış bir uygulama **öz,kısa(concise) klasörlere** **(broken down into)ayrılacak,bölünecek** olsa da, compoenentiniz için ayrı bir dosya oluşturmak, index.js dosyaları denizinde kaybolmaktan kaçınmanıza yardımcı olabilir.

* **Our First Storybook Stories(sayfa 32)**

Artık bu yeni componentleri çalışırken görmeye hazırız. Uygulama ekranlarımızı aktif olarak geliştirirken componentleri içerecek şekilde güncellemek yerine, workspace olarak hizmet edecek bir story ekleyeceğiz.

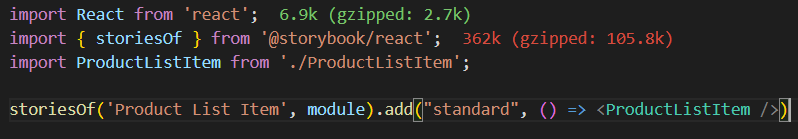
**Storybook story’leri oluşturmanın birkaç yolu vardır**. Her birinin kendi **yararları(benefits)** ve **trade-offs(dengleri)** vardır. Aşağıdaki örneklerde, komut satırından **npm run storybook** (veya thread storybook) çalıştırdıktan sonra **aşağıdaki görüntüye benzeyen bir Storybook workspace** elde edeceğiz. /src/components/ ProductListItem/ProductListItem.stories.js **ProductListItem için bir story oluşturacağız.** Storybook'u kurduğumuzda oluşturulan **global bir stories klasörü** olduğunu fark etmiş olabilirsiniz. Bazı ekipler, hikayelerini ana src dizininin dışındaki bir klasöre yerleştirmeyi tercih eder. **Biz hikayelerimizi bu global hikayeler klasörü yerine temsil ettikleri componentlerin yanına yerleştireceğiz**. Bu, tartışma için kişisel/ekip tercihi bir öğedir, ancak **genellikle hikayeler tanımladıkları componentlerin yakınındayken gezinmenin daha kolay olduğunu görüyorum.**



* **StoriesOf**

**StoriesOf API**, Storybook'ta **stories oluşturmanın orijinal yoludur.**

**StoriesOf**, stories’leri tanımlamak ve **component workspace** de yapılandırılmış bir şekilde oluşturmak için Storybook tarafından sağlanan özel bir fonksiyonu kullanır. Örneğimiz için, **StoriesOf** yöntemlerinin kullanımı aşağıdaki kod örneğine benzeyebilir:



**FluentInterface** nedir? Yazılım mühendisliğinde akıcı bir arayüz, tasarımı büyük ölçüde yöntem zincirlemeye dayanan nesne yönelimli bir API'dir. Amacı, alana özgü bir dil oluşturarak kodun okunabilirliğini artırmaktır.

**Method chaining:** aynı object’in başka bir methodunda bir methodu çağırma mekanizmasıdır.

Örn;

var recipeObj = new Recipe();

recipeObj.addIngredient('salt');

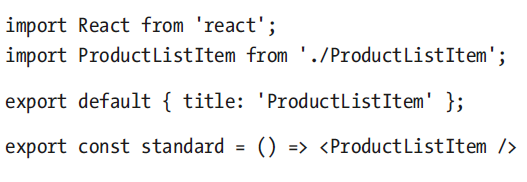
recipeObj.addIngredient('pepper');

**StoriesOf API**, stori’lerimizi tanımlamak için **method chaining** kullanan bir **FluentInterface** veya a mechanism’dir. **StoryOf** yöntemindeki story group ve zincirleme **add methodu** takip eden storyleri sağlıyoruz. Add methofları üzerinde çalışmaya devam edebiliriz, ancak şu anda componentlerimiz yalnızca bir durumunu temsil ediyoruz. İlk parametre olarak **story** adını ve ikinci parametre olarak **story** method veriyoruz. Story method standart bir fonksiyon olabilir, ancak arrow fonksiyon biraz daha okunaklı buluyorum.

**.add() ilk parametresi story ismi, ikinci parametresi story methodları alır.**

* **Component Story Format**

2019'da Storybook ekibi, Component Story Format adı verilen stor’es oluşturmak için başka bir strateji yayınladı. **Component Story Format”i** çoğunlukla **deyimsel JavaScript(idiomatic JavaScript)** kullanarak öyküler oluşturmanın kısa ve öz bir yoludur. **StoriesOf API, stories tanımlamak için özel, Storybook-specific interface kullanırken, Component Story Format JavaScript objects ve functions kullanır:**



Component Story Format bu kullanım tercih edilecek.

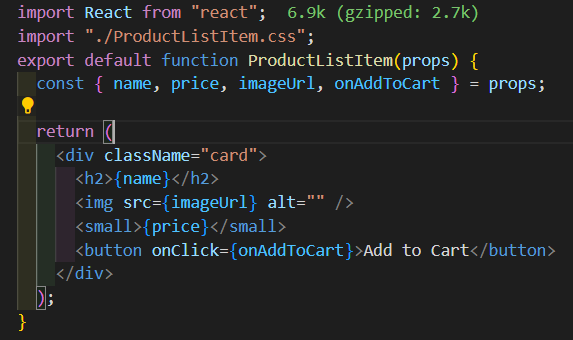
* **Component Code**

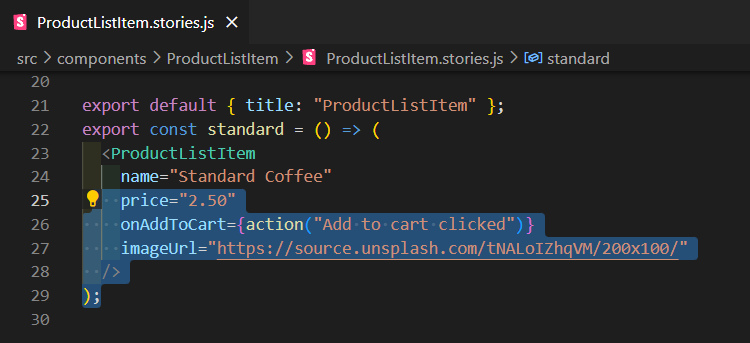
Componentlerimiz ne yapması gerektiğini düşünerek başlayacağız.

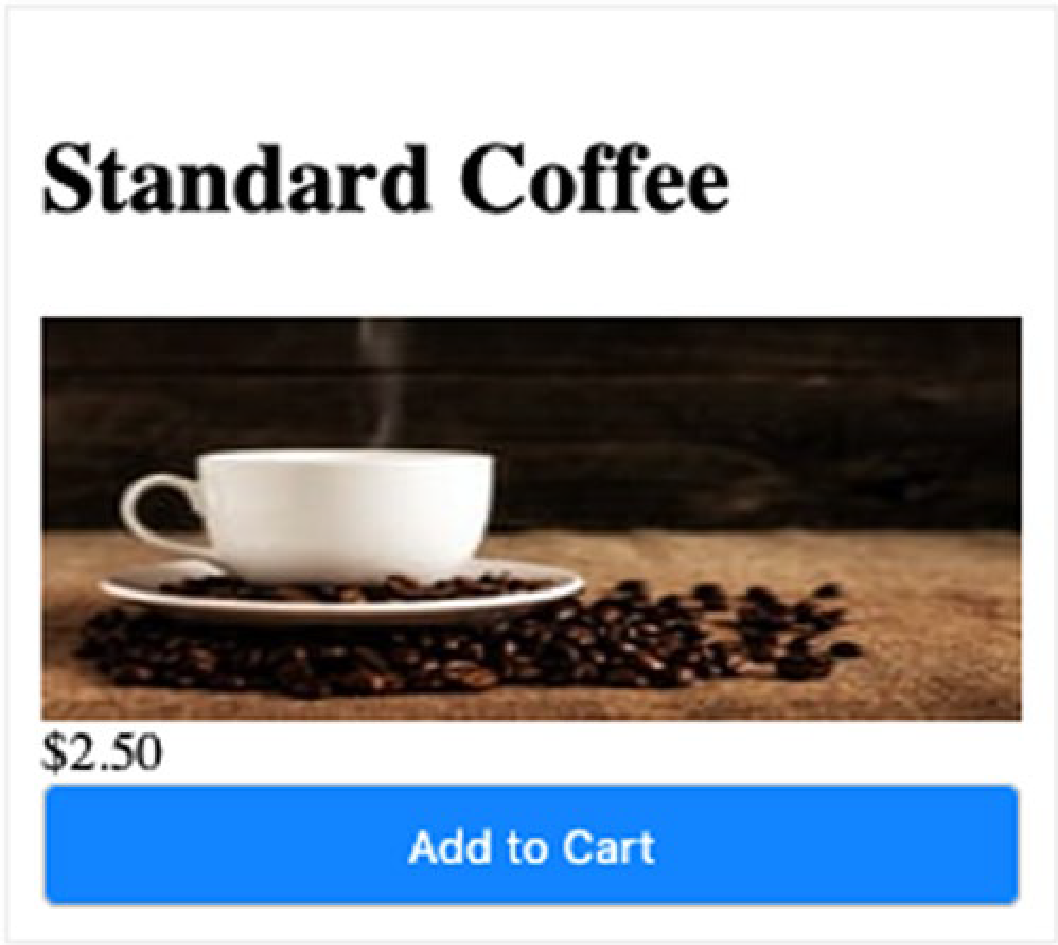
Bu ProductListItem product name, image, price görüntülemekten sorumludur ve bir fonksiyon müşteri(shopper) ‘Add to Cart.’ tıkladığında aktive olacak.

Kodda bunu, bileşenin aldığı name, price, image ve onAddToCart olarak React props'larına çevireceğiz. Örnek componentimiz aşağıdaki kod çıktısı gibi görünebilir. Şu anda Storybook çalıştırmıyorsanız, şimdi npm run storybook (veya yarn storybook) çalıştırmak için iyi bir zaman olacaktır.

We’ll update ProductListItem.js to the following:



Sonra da ProductListItem.stories.js’e de ekleme yaptık.

 Görüntü bu çıktı.

* **Storybook Add-ons (Eklentiler)**

We’re going to use Storybook’s *actions* add-on to remove our log statement in the story while still seeing when our event runs.

We’ll start by importing “@storybook/addon-actions” and referencing the “action” method instead of “console.log.” We’ll pass the label we want to apply to the callback as the parameter to this method, “Add to cart clicked.” When we’re done, our updated story for *ProductListItem.stories.js* should look like the following (only relevant updates displayed):

import { action } from '@storybook/addon-actions';

export const standard = () => (

<ProductListItem

...

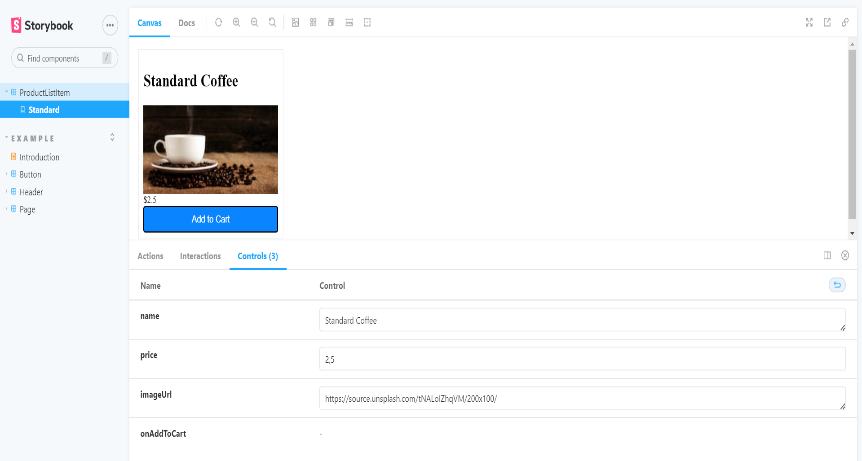
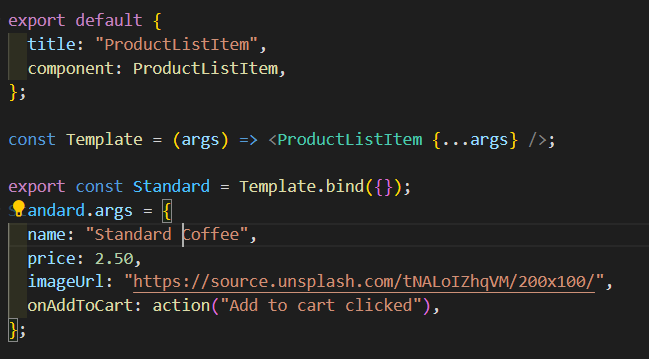
onAddToCart={action("Add to cart clicked")}

...

);

Şimdi, comonentlerin her biri için yeni bir story hardcode etmeden farklı isimler, fiyatlar ve diğer parametrelerle görmek istersek ne olur?

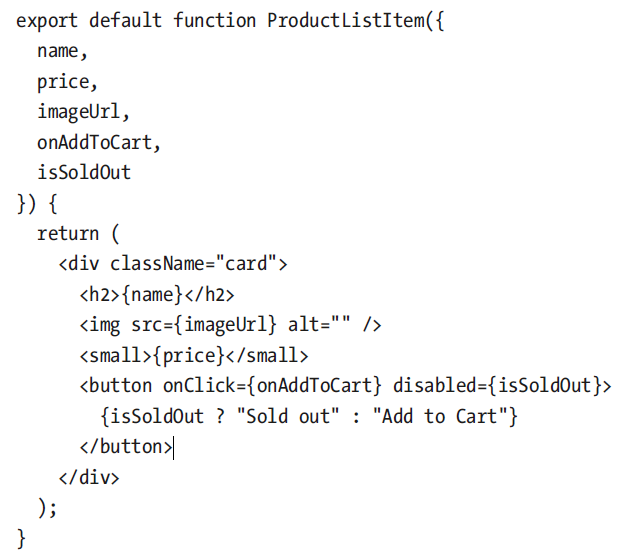
Bu, Knobs Storybook eklentisi öneriyodu fakat **deprecating**(kullanımdan kaldırmak) @storybook/addon-knobs in favor of @storybook/addon-control



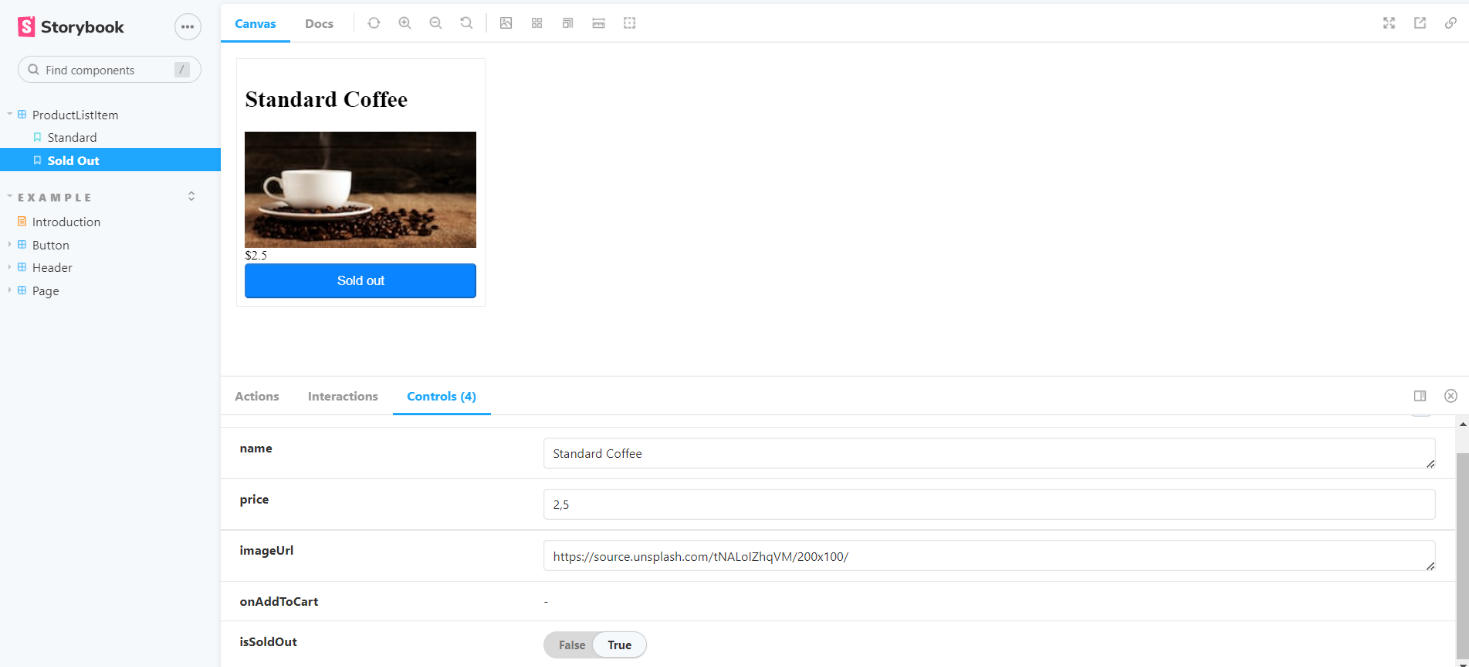
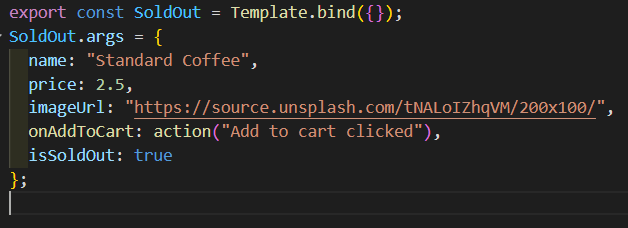
* **Story Variants**

Bir **item(öğenin**) tükendiğini belirtmek gibi, component’lerimizin farklı **global state’lerini** nasıl temsil etmeliyiz? Aşağıdaki örnekte, **ProductListItem componentimize** **isSoldOut** adlı bir **prop** eklediğimizi (**assume)varsayalım**. Bu **prop**, "Sepete Ekle" düğmesinin etkinleştirilip etkinleştirilmediğini ve düğmenin içinde hangi metnin görüntüleneceğini belirleyecektir:

ProductListItem componentine eklenen kod;



ProductListItem .storires.js eklenen kod;



* **When Should We Create a Story vs. Krops ?**
* **Key Takeaways (Chapter 3 için Önemli Bilgiler)**

Bu bölümde, ilk Storybook stories oluşturduk ve bu kitabın büyük bir bölümünde üzerinde çalışacağımız front-end application tartıştık. Stotylerimizi oluşturabileceğimiz birkaç farklı yoldan ve hikayelerimizi oluşturmak için **Component Story Format’ınnı** neden kullanacağımızdan bahsettik. Son olarak, Knobs aracılığıyla hikayelerimize nasıl çeşitler sunabileceğimizi ve yeni stories yaratmayı ve bir stratejiyi diğerine tercih etmenin bazı nedenlerini konuştuk.

**-Chapter 4 Creating Reusable Components(47-57)**

Bu component çalışırken, yeniden **kullanılabilirliği(reusable)** ve **değişime karşı dayanıklılığı( resiliency to change)** sağlayan stratejiler kullanmaya başlamak istiyoruz.

* **What Makes a Good Component?**

Front-end ekosisteminde, bir componentin nasıl **yapılandırılması(structured)** gerektiğine dair birçok görüş vardır. Product List örneğimizi nasıl yapılandırmamız gerektiği konusunda topluluğa anket yaparsak, çelişkili öneriler alabiliriz. Bazıları, all the product list item’ların **concerns(endişeleri)** için tek bir component yapmamız gerektiğini söyleyebilir. Diğerleri, her HTML element için yeni bir component oluşturmayı önerebilir.

Bu stratejilerden herhangi biri işe yarayacaktır, ancak iyi bir component’in neyin oluşturduğunda genellikle nüanslar olduğunu görüyorum

* **All Comes Down to Purpose(Amaç)**

export default function ProductListItem({ name, price, imageUrl, onAddToCart }) {

return (

<div className="card">

<h2>{name}</h2>

<img src={imageUrl} alt="" />

<small>{price}</small>

<button onClick={onAddToCart}>Add to Cart</button>

</div> );

}

Many areas of our codebase will need ways to present content within a **consistent(tutarlı) card**, **display text and headings**, **handle click/press interactions**, and so on.Bu componenti bir dizi **reusable compone**nt’e dönüştürelim.

* **Additional Components(Ek componentler)**

**Reusable elementleri kendi componenleriyle sarmalayacak(wrapped)** şekilde hareket ettirerek başlayacağız. Bunlar şimdilik çoğunlukla **pass-through components** olacak - yani, we’ll simply p**ass props to the JSX elements.** Bu kitap boyunca ilerlerken, bu **ayrımdan(separation)** yararlanarak bu component’lere bazı powerful **functionality’ler** uygulayacağız:

function Heading({ children }) {

return <h2>{children}</h2>;

}

function Card({ children, highlight }) {

const cardClassName = highlight ? "card sale" : "card";

return <div className={cardClassName}>{children}</div>;

}

function Text({ children }) {

return <span>{children}</span>;

}

function Button({ onClick, children }) {

return <button onClick={onClick}>{children}</button>;

}

**Var olan ProductListItem.js componentimizi alttaki gibi güncellicez.**

export default function ProductListItem({name, price, imageUrl, onAddToCart, isSoldOut, isOnSale,}) {

return (

<Card highlight={isOnSale}>

<Heading>

{name} {isOnSale && "(On Sale)"}

</Heading>

<img src={imageUrl} alt="" />

<Text>{price}</Text>

<Button onClick={onAddToCart} disabled={isSoldOut}>

{isSoldOut ? "Sold out" : "Add to Cart"}

</Button>

</Card> );

}

* **When Should We Abstract(Soyutlama) Components?**

**This process of removing some elements from one component into a new component can be referred to as abstracting a component.** Yeni bir component olabilecek herhangi bir elementten basitçe yeni bir component oluşturmakla ilgili bir sorun var. Ne zaman bir **abstract** yaratmamız gerektiğini belirlemek önemlidir.

Yeni bir componentin yararlı olup olmayacağını belirlemek için kullandığım birincil faktör, **bir kod tabanında aynı amaca hizmet eden bir componente sahip olan veya ihtiyaç duyacak başka componentlerin olup olmadığıdır.** Basit örneğimizde, Rocket Coffee için oluşturulmuş diğer bileşenlerin bir Card componentlerine ihtiyaç duyacağını varsayabiliriz. Bir elementin yeniden kullanılacağından emin değilsek, bir **abstract** oluşturmadan önce beklemek daha güvenlidir.

* **Component States**

Birçok **component**, farklı durumları temsil etme yeteneğini gerektirir. product list yüklenirken veya yükleme başarısız olduğunda bir durumu temsil etmemiz gerekebilir. /src/components içindeki ProductListItem klasörünün yanında bir ProductList klasörü oluşturalım. **Component yüklenirken ve bir hata oluştuğunda bir duruma(state)** sahip olmak istiyoruz. Bu durumları Boolean bayraklarıyla aşağıdaki gibi gösterebiliriz:

export default function ProductList ({ isLoading, hasError, ...otherProps }) {

if(isLoading) {

return <Loading />;

}

if(hasError) {

return <Error />;

}

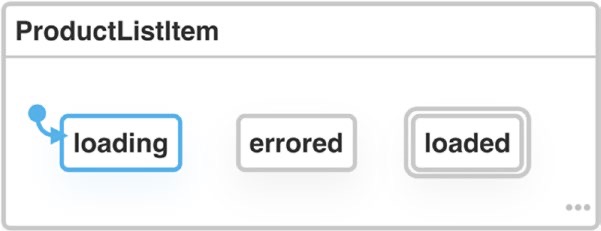
return // standard output when data present

}

Bu kod işe yarayabilir, ancak hatalara yol açabilir. Teknik olarak yükleme durumundaysak ve bir hata oluştuysa ne olur? Hem **isLoading** hem de **isErrred'ın** **true olarak ayarlanmasıyla sonuçlanabiliri**z. **Componentlerimiz asla hem yükleniyor hem de bir hata oluşmuş durumda olmamalıdır.** Uygulamamızı kullanan herkes için, bu, verilerin yüklenirken bir hata olduğuna dair bir not sunulması gerektiğinde, verilerin sürekli olarak yükleniyor gibi görünecektir.

Hata sınırları ve bileşen hiyerarşisinde daha üst sıralarda yer alan hataların ele alınması gibi birçok yolu olsa da, **component’lerimizin yalnızca tek bir doğru durumda(state de) olmasını sağlamaya çalışmalıyız.**

**the machine can have different states, but at a given time fulfills(yerine getirmek) only one of them.(Machine birden fazla state olabilir ama belirli bir zamanda bunlardan sadece birini yerine getirebilir.)**



*visualization of our simple state machine*

Bu felsefeyle, basit örneğimizi güncelleyebiliriz. Component statelerimizi içerecek bir statusTypes nesnesi ekleyerek başlayacağız:

export const statusTypes = {

loading: "loading",

errored: "errored",

loaded: "loaded"

};

This object contains all the states that our component can be in. These values could be any other string, but using the same property name and value should work well enough for most instances:

**Aşağıdaki kodda status prop gelir ve farklı statusTypes birine eşleşir. Tek birini gösterir. Fakat yukarıdaki kodda isLoading, hasError ikiside true gelse ikisinin durumu göstermesi hatalı olacaktı.**

export default function ProductList({status, ...otherProps}) {

if (status === statusTypes.loading) {

return <Loading />;

}

if (status === statusTypes.errored) {

return <Error message="Failed to load data" />;

}

return ... // standard output when data present

}

Ardından, **various(çeşitli) component stateleri** çalışırken görmek için bazı hikayeler oluşturabiliriz. ProductList.stories.js'yi oluşturacağız.

* **How Should We Arrange(Düzenlemek) Our Components?**

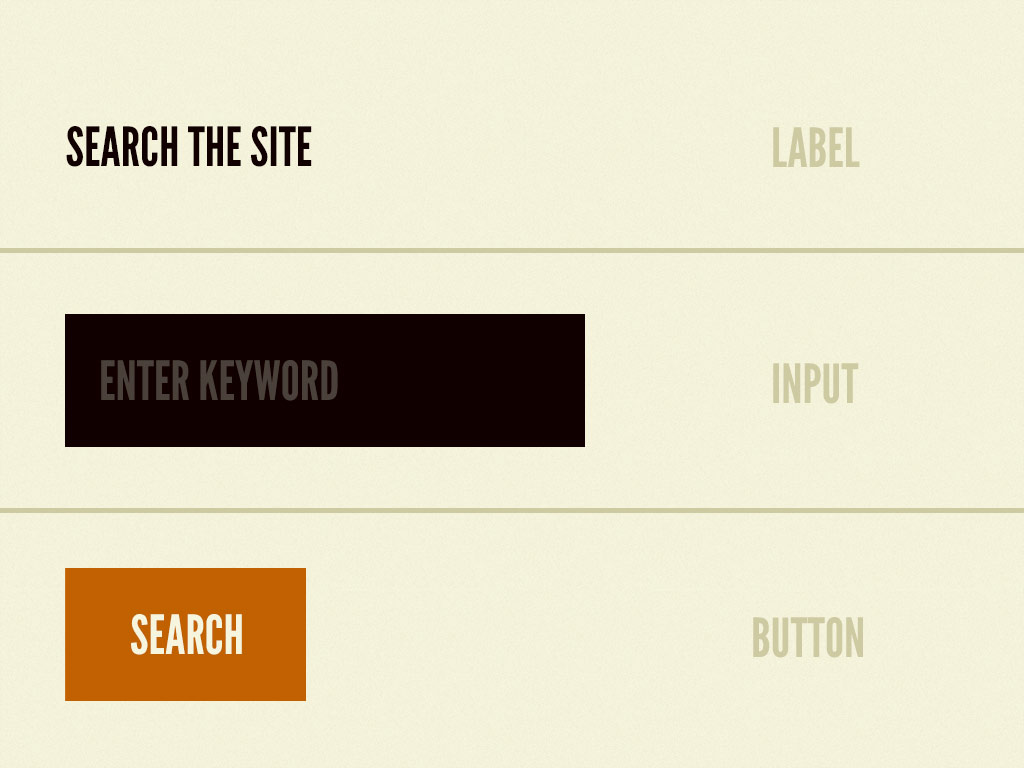
Varsayılan olarak, library olan React, component’lerin kod tabanımızda(codebase) nerede olması gerektiği konusunda çok az fikre sahiptir. B**azı kütüphaneler ve frameworkler, şeylerin nereye ait olduğu konusunda çok fazla fikir sahibi olsa da, React herhangi bir güçlü öneri sunmuyor. Bir component library oluştururken bulduğum, takip etmeyi sevdiğim birkaç yönerge(guidelines) var.**

**1.Classifying Component Types**

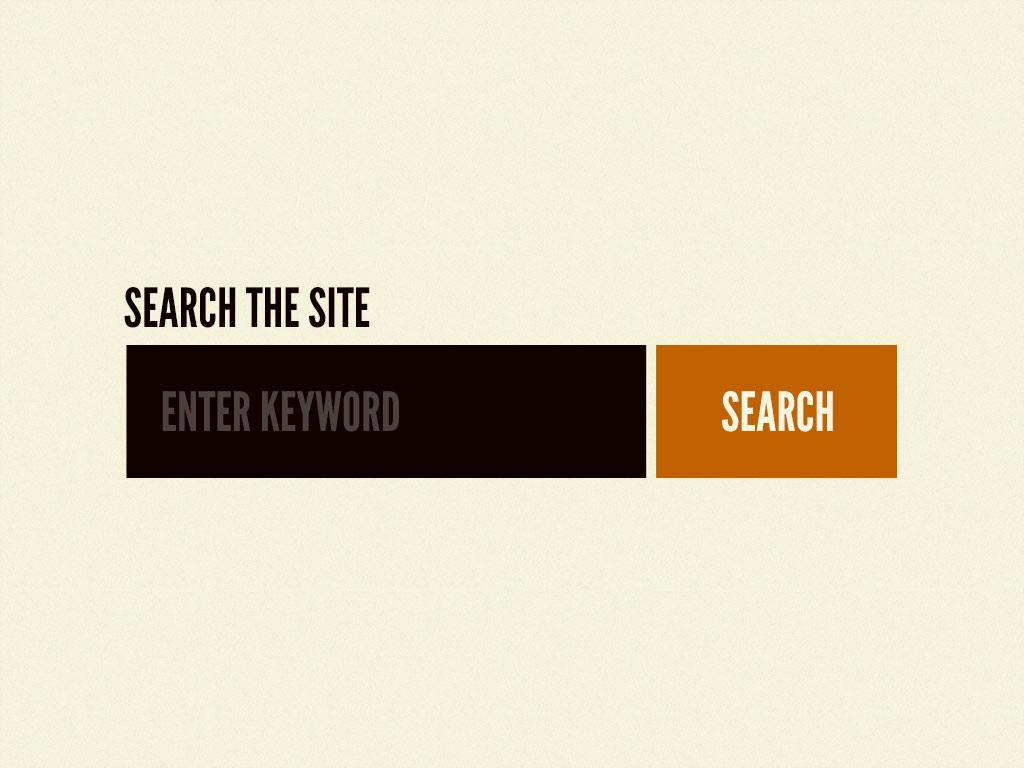
One **guideline(kılavuz**) I like to follow is grouping components together. There’s a problem here—we don’t **necessarily(muhakkak, şart)** have a great metric to ensure that we’re **classifying(sınıflandırmak) components** the same way.

**Atomic Design Nedir? :** There are five distinct levels in atomic design:

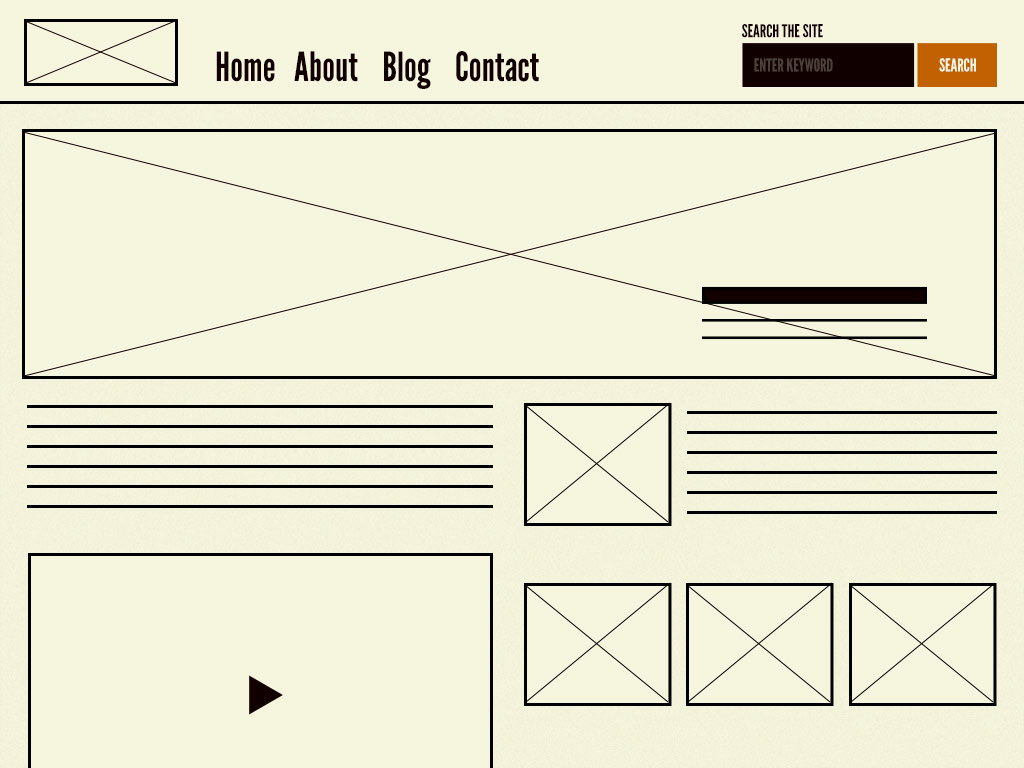
1. [**Atoms**](https://bradfrost.com/blog/post/atomic-web-design/#atoms)**:** Web interfaces uygulanan atomlar, form label, input veya button gibi HTML tags’lardır.



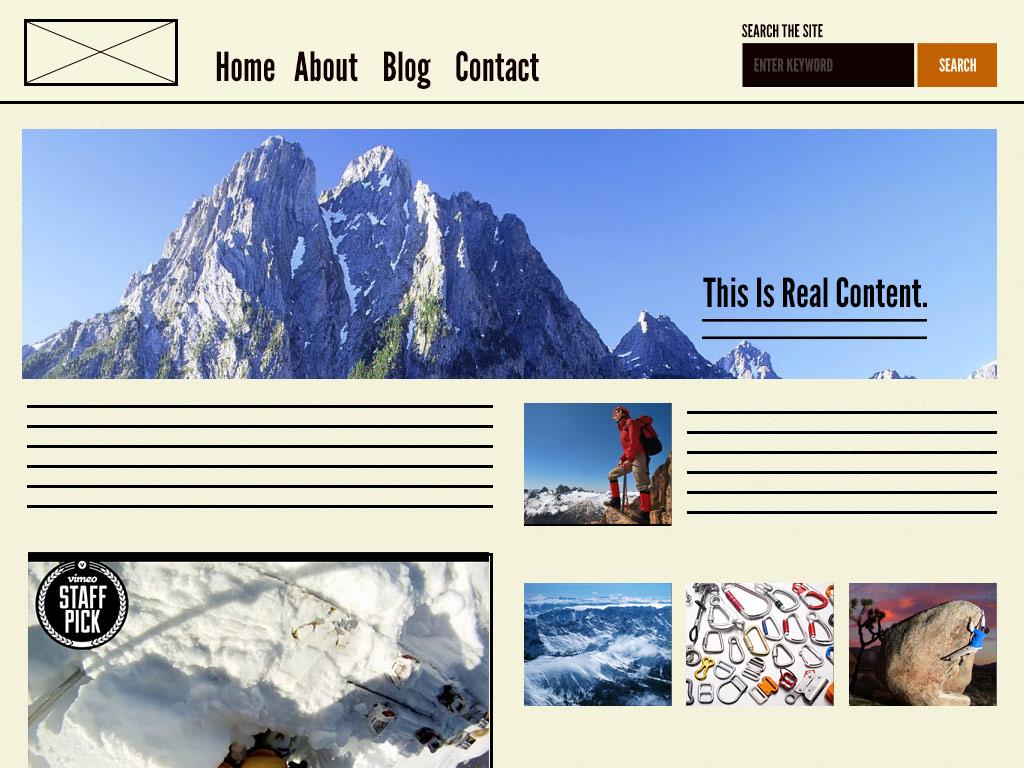
1. [**Molecules**](https://bradfrost.com/blog/post/atomic-web-design/#molecules)**:** For example, a form label, input or button aren’t too useful by themselves, but combine them together as a form and now they can actually do something together.



1. [**Organisms**](https://bradfrost.com/blog/post/atomic-web-design/#organisms)**:** 
2. [**Templates**](https://bradfrost.com/blog/post/atomic-web-design/#templates)**:**



1. [**Pages**](https://bradfrost.com/blog/post/atomic-web-design/#pages)**:**



Brad Frost'un **Atomic Design** adlı harika bir kitabı (https://atomicdesign.bradfrost.com/) ve makalesi (https://bradfrost.com/blog/post/atomic-web-design/) var. **Atomik Tasarımda Brad, UI öğelerini periyodik element tablosuna dayalı olarak classifying(sınıflandırmak) için bir mekanizma tanımlar.** Bu felsefede, en düşük seviyedeki componentlerin **atomlar olduğu, moleküller, organizmalar** vb. şeklinde oluşturulabilen **building design systems out of components references(atıfta bulunur)**. Başka bir deyişle, **temel componentler veya atomlar, sonunda uygulama ekranlarımızı(screen) veya sayfalarımızı(page) oluşturan daha büyük componentler halinde bir araya getirilir.** Bu felsefeyi seviyorum ve uygulamalarımızı oluşturan componentler hakkında düşünmenin gerçekten yararlı bir yolu olduğunu düşünüyorum.

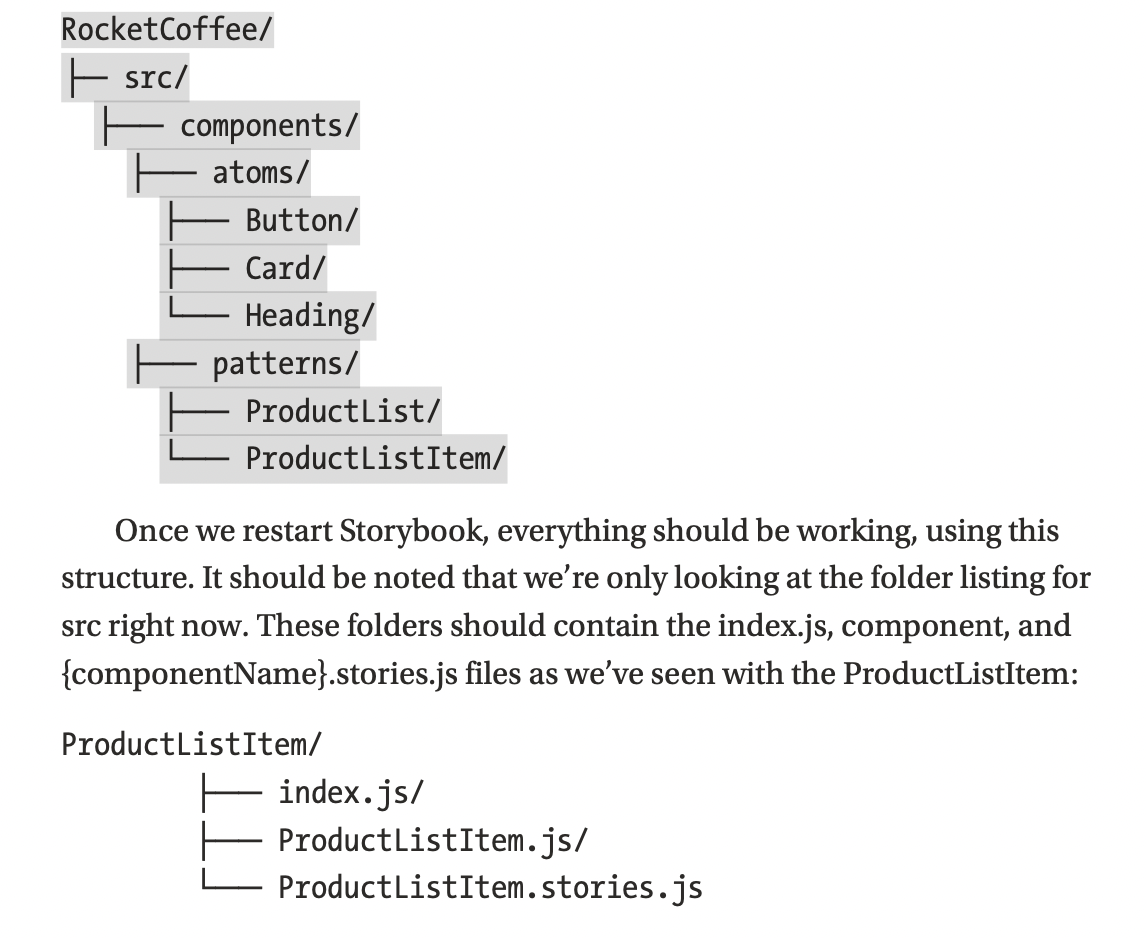
Tüm bunlar, genellikle biraz farklı bir adlandırma şeması kullanıyorum:

**Atoms** – Atomic Design isimlendirmeden esinlenerek, genellikle **foundational(temel) components atoms** diyorum. Bunlar, **buttons, headings, text** ve kendi başlarına pek iyi durmayan, ancak uygulamamızdaki diğer öğelerin başarısı için gerekli olan şeylerdir.

**Patterns** – **Bu componentler, atom düzeyindeki componentler oluşan yeniden kullanılabilir UI kalıplarıdır. ProductListItem componentlerimizi,** **Card, Button, Heading ve Text** **componentlerinden** oluştuğu ve birçok bağlamda değerli olabileceği için bir model olarak sınıflandırırdım.

**Screens** – **Bu componentler, bir uygulamadaki sayfalarımızı veya ekranlarımızı temsil eder.** **Screen**, kullanıcının göreceği **product listing page** gibi bir şey olabilir. Bu, bir **layout, product list** ve diğer çeşitli öğelerden **consist(oluşabilir)**. **Şu anda genel olarak kullandığım adlandırma şeması bu olsa da**, **bunun bir dizi kesin kural olmadığını belirtmek önemlidir.** Daha da önemlisi, **sizin veya ekibiniz için uygun olan tutarlı adlandırma/sınıflandırma(naming/classification) kullanmaktır.**

Bunu akılda tutarak, **ProductList.js ve ProductListItem.js** dosyalarımızı **components/patterns** olacak şekilde güncelleyebiliriz. Bu sırada, ana ProductListItem'den taşıdığımız diğer bileşenlerle eşleşen yeni klasörler oluşturmalıyız. component (ancak ProductListItem.js dosyasında tutulur). İşimiz bittiğinde, aşağıdaki gibi bir dosya yapısı elde etmeliyiz:



* **Key Takeaways (Önemli Çıkarımlar)**

Bu bölümde, **iyi scale(ölçeklenebilen) componentlerin nasıl yazılacağını** tartıştık. Unix felsefesinden ilham alarak tek bir şeye odaklanan ve o şeyi iyi işleyen componentler oluşturduk. **Bir componentin amacının nasıl net olması gerektiğinden bahsettik** ve bu amaca odaklanmak, hangi items benzersiz componentlerin olarak var olması gerektiğini aydınlatmaya yardımcı olabilir. Finally, we talked about components as a series of states and some strategies we can use to classify(sınıflandırma) our components.

**-Chapter 5 Styling(59-72)**

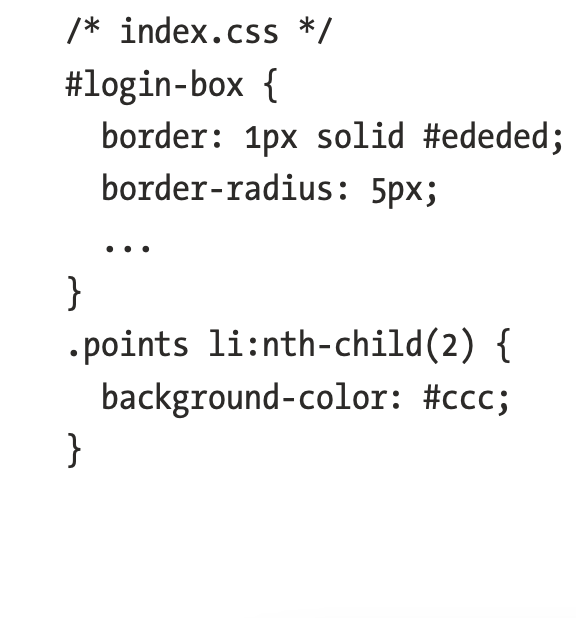
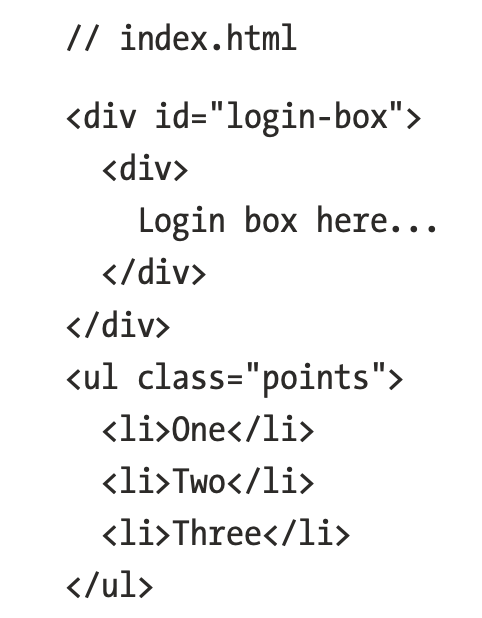
Styling uygulamaları, diğer kodlama paradigmalarının sahip olmadığı benzersiz zorluklara sahiptir, ancak kodumuzun çok az alanı, kodumuzu nasıl biçimlendirdiğimiz kadar hızlı bir etki yaratır.

Styling landscape gezinmek zor olabilir, ancak olması gerekmiyor. Bu bölümde, sahip olduğumuz seçeneklerden bazılarını ve nerede birini diğerine tercih etmek isteyebileceğimizi tartışacağız. Çoğunlukla uygulamalarımızı nasıl şekillendireceğimizi ve stillerimizin uygulamalarımızla **scale(ölçeklenmesini)** sağlamak için yerleştirmek isteyebileceğimiz **structure(yapıyı)** seçmeye odaklanacağız. **Bu bölüm, uygulamalarımızı şekillendirmenin syntax üzerine derin bir dalış olmayacak.**

* **CSS**

Web uygulamalarının **stillendirilmesiyle(styling)** ilgili herhangi bir tartışma **CSS**

**(Cascading Style Sheets)(Basamaklı Stil Sayfaları)** ile başlamalıdır. CSS, web uygulamalarına stil vermenin tarayıcı destekli yoludur. CSS kullanarak, HTML elementlerine nitelikler uygularız ve bu elementlere uygulanması gereken stilleri **(selectors)seçiciler** aracılığıyla bunlara başvurarak tanımlarız. **Selector**, **bir HTML elementine refer(bahsetmenin)** **bir yoludur.** Bazı durumlarda bu, bir elemente bir **className, prop,** veya **bir id niteliği uygulamak** veya elementi başka yollarla bulmak yoluyla olabilir:



**login-box’ı elementini id ile seçiyoruz. Element id benzersiz olmalıdır - bir tanımlayıcı(identifier) ile birden çok öğe seçmek istiyorsak bir class kullanabiliriz.**

Ayrıca **nth-child pseudo class’ı(sözde sınıfı)** kullanarak listedeki **ikinci element’e** bir **background color** uyguluyoruz. Bu, bir gruptaki sırasına göre tek bir öğe seçmemizi sağlar. Bu örnek, CSS ile yapabileceklerimizin yalnızca yüzeyini scratches(çiziyor). CSS hakkında daha ayrıntılı bir araştırma için, Martine Dowden ve Michael Dowden'ın Architecting CSS: The Programmer's Guide to Etkili Stil Sayfalarına bakmak isteyebilirsiniz.

* **Benefits of CSS (CSS’in Faydaları)**

CSS, uygulama geliştirirken **(encounter)karşılaşacağımız** hemen her tarayıcıda **(interpreted)yorumlanabilir.** Bir uygulamayı, ön-derleyici**(preprocessors)** vb. biçimlendirebileceğimiz diğer yolların çoğu, bir tarayıcıda çalışmak için bir **(translation)çeviri veya (compilation)derleme** adımı gerektirir. **Varsayılan olarak, CSS'nin böyle bir çeviriye ihtiyacı yoktur.** Ek olarak, farklı geliştiricilerin CSS'yi nasıl kullandığı konusunda nüans olsa da, bu her **(ubiquitous paradigm)yerde bulunan bir paradigmadır**. Birçok web geliştiricisi, daha specialized styling library yabancı bir kavram olabileceği CSS ile biraz deneyime sahiptir.

* **Drawbacks of CSS (CSS’in Dezavantajları)**

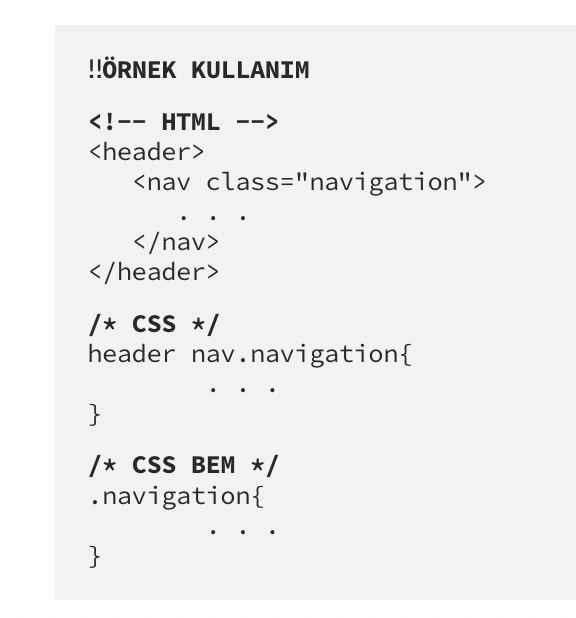
Bu başlık kasıtlı olarak misleading(yanıltıcıdır). Farklı stil teknolojilerinin pros(artıları) ve cons(eksileri) hakkında konuşurken, CSS ile sürekli olarak ortaya çıkan öğelerden biri cascading (CSS'deki C). **Cascading, özelliklerin diğer özelliklerin üzerine yazmasına izin veren CSS'nin özelliğidir.** Bu, bu özelliğin farkında olmayan bazı geliştiriciler için kafa karıştırıcı olabilir - özellikle stil oluşturmayı sonradan düşünülen kod tabanlarında.

**Cascading** CSS'nin faydalarından biri olduğu kesinlikle tartışılabilir olsa da, **CSS ile rahat olmayanların uygulamaları şekillendirirken takılabileceği bir şey olduğundan, bunu bir dezavantaj olarak da tartışıyoruz.** . Eklediğiniz stillerin uygulanmadığını bulmak için kod tabanının bir alanındaki stili değiştirmeyi hiç denediniz mi? Uygulamanızdaki başka bir şeyin stillerinizi geçersiz kılıyor olması muhtemeldir. **CSS ile interpreted(yorumlanan) son özellik, uygulanan son özelliktir.**

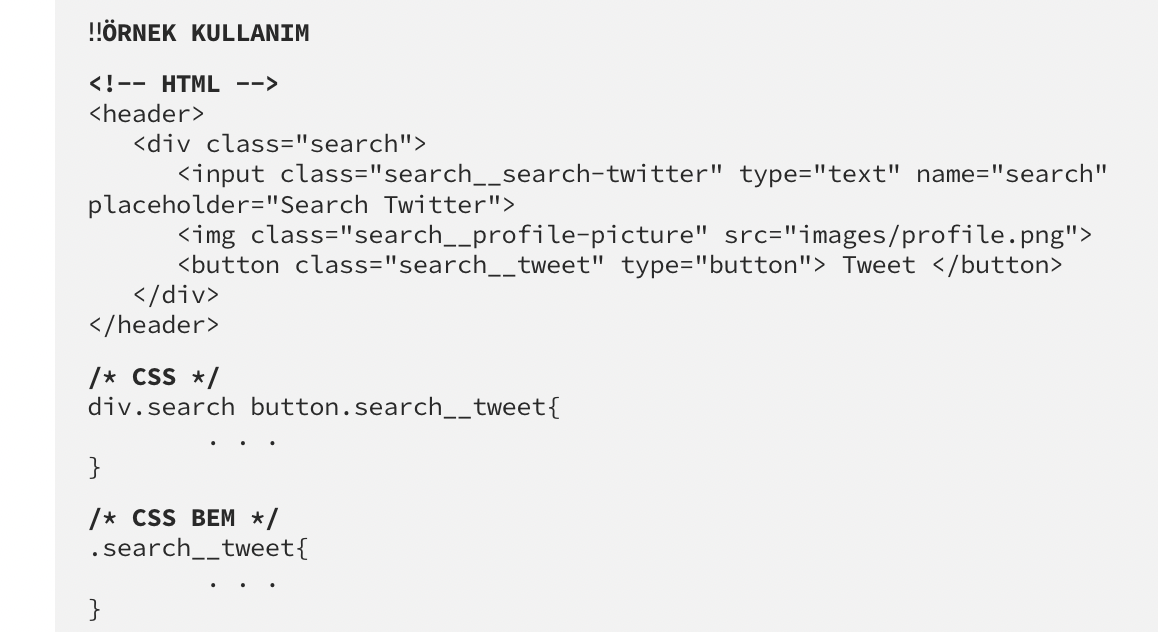
CSS selectors çalışma şekli **(duo to)nedeniyle**, adlandırma çakışmaları olması veya stil vermek (intending)istemediğimiz elementlere stillerin yanlış uygulanması kolay olabilir. **Çakışmaları(conflicts) önlemek için dikkatli adlandırma ve kod yapısı(code structure) uygulanmalıdır.**

**\*Note: Element specificity() ve çakışmalardan kaçınmaya yardımcı olabilecek BeM (Block element Modifier) adlı bir teknik olduğunu unutmayın.** BeM hakkında daha fazla bilgiyi **getbem.com** adresinden edinebilirsiniz.

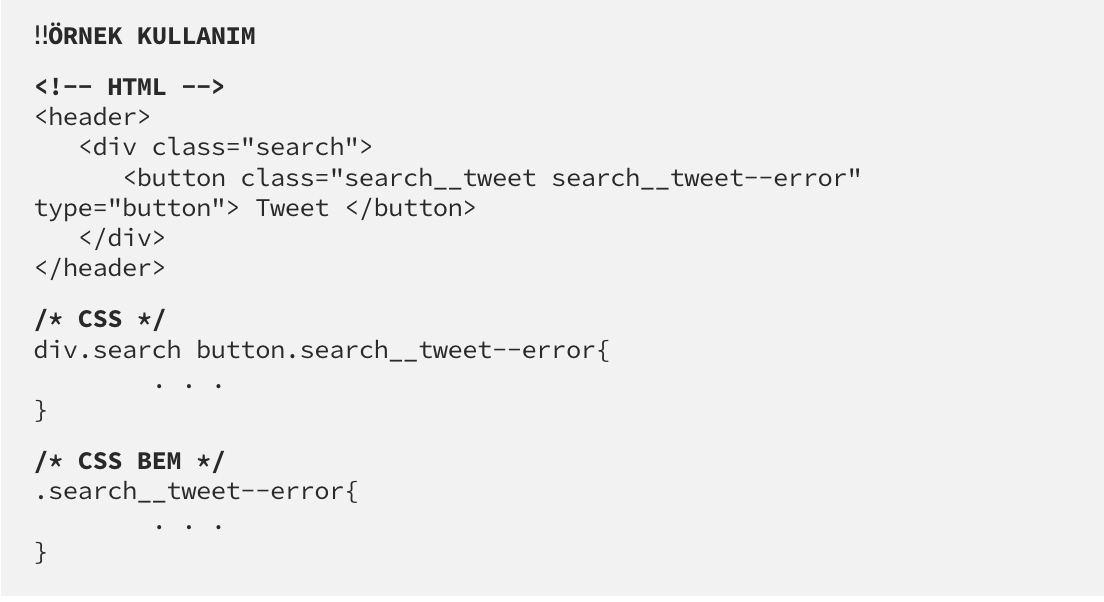
**BeM örnek;**

-Kuralı: Element adı block adından \_\_ ile ayrılır (*.block\_\_element*).

**Block:** Blok adı genelde .navigation gibi tek bir kelimedir, ancak daha uzun blok tanımınız varsa tek bir tire ile bölerek yapabilirsiniz(.left-side gibi).



**Element:**.search block içerisinde search-twitter element, profile-picture element, tweet element kısımlarından oluşuyor. (.search\_\_search-twitter) gibi (alt çizgi 2 tane)

**Modifier: ( .search\_\_tweet--error ) tire’den iki tane** 

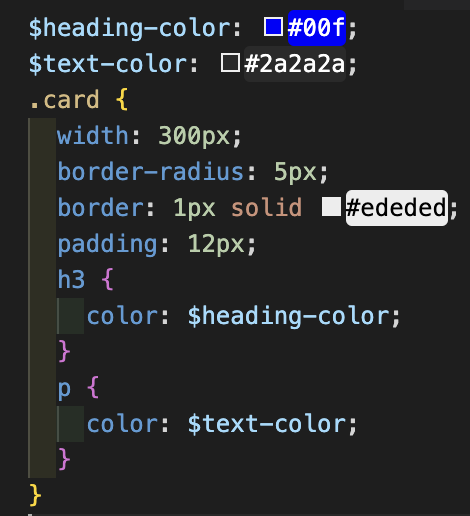
-İç içe olan kullanımlarda *\_\_* ile isimlendirme yapsakta birden fazla olduğu durumlarda bir *\_\_* daha ekleyip isimlendirme yapmak yanlış olur.

Doğrusu budur;



* **Preprocessors ( ön-derleyici)**

Bir **CSS preprocessor** adından da anlaşılacağı gibi, **farklı stil sözdizimini işleyen ve tarayıcının anlayacağı CSS'ye dönüştüren bir şeydir. Yaygın bir preprocessor Sass'tır.** Sass/SCSS sözdiziminin bir örneği aşağıdaki gibidir:



Alttaki html karşılığı;

<div class="card">

<h3>This is the heading</h3>

<p>This is some text</p>

</div>

İlk bakışta bu kod tıpkı CSS gibi görünebilir. Bununla birlikte, daha fazla inceleme yapıldığında, Sass/SCSS ve standart CSS'nin oldukça farklı olduğu görülebilir. **Sass, benzer değişkenler ve iç içe(nested) element stilleri için ek işlevsellik sağlar.** Bu sağlam bir seçenek olabilir, ancak CSS'nin custom properties (veya **değişkenleri(variablles)**) desteklediğine dikkat edilmelidir. **Ek olarak, bu kitap yazılırken CSS Çalışma Grubu CSS'ye yerleştirmenin eklenmesini tartışıyor**

**Not:** Sass/SCSS ve diğer **CSS** **preprocessor**, burada tartışılan öğelerin ötesinde birçok yararlı işlevsellik(functionality) ekleyebilir. Bununla birlikte, bireysel özellikler bu kitabın kapsamı dışındadır.

* **Benefits of Preprocessors (Preprocessors faydaları )**

**Preprocessor** CSS'ye büyük functionality ekleyebilir. Bir **Preprocessor** kullanmak, stil kodumuza ek yardımcı programlar sağlayarak, uygulamamızın mantığını nasıl yazdığımıza daha yakın hissedebilecek bir şekilde stil kodu yazmamıza olanak tanır. **Preprocessors** kilidini açtığı bu tür yeteneklerden bazıları **variables, conditional statements, loops, and property nesting.**

* **Drawbacks of Preprocessors (Preprocessors Dezavantajları)**

**Preprocessors**, **dışa aktarılan(exported) kodumuza zaman ve hatta dosya boyutu ekleyebilecek ekstra derleme(build) adımları ekleyebilir.** Ek olarak, **Preprocessors** syntax, geliştiricilerin öğrenmesi gerekebilecek yeni kodlar ekleyebilir.

* **CSS-in-JS (64)**

JavaScript'te stilleri işleme(handling styles) potansiyel olarak bu kitaptaki en **tartışmalı(controversial)** konulardan biridir. Bazı geliştiriciler, JavaScript'e front-end sorumluluğunu daha fazla zorlamayı severken, diğerleri bunu yüksek **bir suç(crime)** olarak görüyor.

CSS-in-JS teknikleri, genellikle **React** ile birlikte gelen “**Her şey bir component’dir”** zihniyetini zorladıklarından, birçok React kod tabanı için popüler bir seçimdir. **Where other frameworks promote separation of concerns, React components layer styling, markup, and functionality as one cohesive unit of code.**(Diğer frameworklerin **separation of concerns** desteklediği durumlarda, React bileşenleri stili, işaretlemeyi ve işlevselliği tek bir birleşik kod birimi olarak katmanlar.)CSS-in-JS kullanmak, stilleri component’in sorumluluğu olarak ele aldığı için bu konsepti biraz daha ileriye götürür.**JS'de CSS'den yararlanmanın bir yolu, React component stili desteğini kullanmaktır:**

const Card = ( ) => {

return (

<div style={{

backgroundColor: "#EDEDED",

borderRadius: 5,

padding: 12

}} >

...

</div>

)}

Önceki card örneğimizde, background-color, border-radius, and padding sağlıyoruz. **Tireler JavaScript özellikleri olarak geçerli olmadığından, geleneksel CSS özellik adları yerine camel case kullandığımıza dikkat edin**. Ayrıca, bu yalnızca bir J**avaScript nesnesi olduğundan, bu stili hareket ettirebiliriz, bu nedenle inline değildir:**

const **cardStyle** = {

backgroundColor: "#EDEDED",

...

}

return (<div style={**cardStyle**} ...);

React'in stil desteğine ek olarak, React ekosisteminde **emotion** ve **styled-components** gibi stil için mekanizmalar sağlayan başka **library** de vardır. **styled-component’lerini** kullanarak, uygulamamızı **tagged** **template literals** ve **styled’s API'**si ile stillendirebiliriz. Bir Card componenti oluşturmaya başlayacağız:

const Card = styled.div`

width: 300px;

border-radius: 5px;

border: 1px solid #EDEDED;

padding: 12px;

h3 { color: ${(props) => props.headingColor}; }

p{ color: ${(props) => props.textColor} }

`;

Bundan sonra**, default props** sağlayacağız. **Card bir React componenti olduğundan, diğer React componentlerine** yaptığımız gibi **defaultProps** eklemek **için aynı API'yi** kullanacağız:

**Card.defaultProps** = {

headingColor: "#00F",

textColor: "#2A2A2A"

}

Finally, we can render our **Card component** similar to how we would any other component:

export default function App() {

return (

**<Card>**

<h3>This is a card heading</h3>

<p>This is the text description</p>

**</Card>** );

}

**Note:** We’re only scratching the surface of what we can achieve using **styled-components** or **emotion**. Both libraries have a mechanism for themes that provides a clear path for providing consistent styles to varying components. **Additionally, there are entire ecosystems of plugins that these libraries can utilize such as styled- system, rebass, and theme-ui.**

* **Benefits of CSS-in-JS**

**Componentlerimizi** **biçimlendirmek(style)** için **CSS-in-JS** kullanmak, **Componentlerimizi** geliştirmek ve biçimlendirmek için tek bir paradigmaya bağlı kalmamızı sağlar. Component functionality’mizi oluşturmak için kullandığımız aynı özellikleri kullanarak **Componentlerimizi biçimlendirmek inanılmaz derecede yararlı olabilir. Ek olarak, CSS'nin basamaklı(cascading) yönleri konusunda rahat olmayan geliştiriciler, CSS-in-JS kullanarak kendilerini evlerinde daha rahat hissedebilirler.**

* **Drawbacks of CSS-in-JS(Dezavantajı)**

**Preprocessors gibi CSS-in-JS de geliştiricilerin öğrenmesi gereken yeni sözdizimi sunar.** Ek olarak, JavaScript ile stil oluşturma, **tarayıcıda ham CSS kullanarak elde ettiğimiz performans avantajlarından mutlaka yararlanmaz.**

**Performans için kötü olacaktır diyor.**

* **Utility-First Styling Libraries (Stil Yardımcı Kütüphaneler)**

Son yıllarda, **utility-first styling libraries** , **Tailwind** ve **Tachyons** gibi kitaplıklar tarafından popüler hale getirilen uygulama **stil(styling)** seçeneklerinin ön saflarına yükseldi. Geliştiriciler, **utility-first** bir yaklaşım kullanarak, bir kod tabanı boyunca tekrar tekrar kullanılabilen birçok küçük **CSS classes or styling functions** oluşturur. Stilleri doğrudan bir bileşene uygulamak yerine, nasıl stillendirileceğini tanımlamak için bir dizi yardımcı sınıf adına başvuruyoruz. **Tailwind'i kullanarak bir Card bileşenini şu şekilde biçimlendirebiliriz:**

*A Card component using Tailwind*

<div class="rounded p-5 m-5 shadow-lg max-w-md">

<h3 class="text-blue-600 text-xl font-bold">...</h3>

<p class="text-sm text-gray-600">...</p>

</div>

Bu yaklaşım hakkında ne düşünüyorsunuz? Tailwind'i gördüğümde ilk tepkim olumsuz oldu, ancak bazı projelerde Tailwind'i kullandıktan sonra, yavaş yavaş **UI uygulamalarına stil vermenin çok hızlı ve tutarlı bir yolu olduğunu gördüm.** **Utility-first libraries , Bölüm 4'te tartıştığımız "Bir şeyi iyi yapın" felsefesini benimser ve kompozisyonlu UI'ler oluşturmanın güzel bir yolunu sunar.**

* **Benefits of Utility-First Libraries**

Yardımcı program öncelikli kitaplıklardan yararlanmak, tutarlı **UI uygulamalarını hızlı bir şekilde oluşturmamıza olanak tanır.** Çoğu durumda, uygulamamız gereken stiller mevcuttur ve stil yardımcı programlarını kullanmak, tüm componentlerimizin aynı görünüme ve hisse sahip olmasını sağlamamıza yardımcı olabilir.

* **Drawbacks of Utility-First Libraries**

Çoğu zaman, **utility-first styling libraries**, sonuçlarına ulaşmak için hala **CSS preprocessor’lerine bağımlıdır.** Bu, **kullanılmayan yardımcı programları ortadan kaldırabileceği için yararlıdır,** ancak ek bir oluşturma adımı da ekler. **Ayrıca bazı geliştiriciler, bir component veya element stilini belirlemek için birden fazla class name sahip olmaktan hoşlanmazlar.**

* **How to Choose a Styling Solution (Bir Styling Solution Nasıl Seçilir )**

**trade-off:** bir şeyi elde etmek için başka bir şeyden vazgeçmek

Bu seçeneklerden herhangi birini seçerken göz önünde bulundurmamız gereken **trade-offs** vardır. Uygulamanızı şekillendirmek için bir yöntem seçerken devreye giren birçok faktör vardır. **Ne yazık ki, sizin veya ekibinizin uygulamanızı nasıl şekillendirmesi gerektiğini seçmek için herkese uyan tek bir yaklaşım bulmamızın bir yolu yok.**

Bir **styling solution** önerirken sıklıkla değerlendirdiğim şey, **bir ekibin CSS ile JavaScript arasında sahip olabileceği rahatlık düzeyidir.** Ek olarak, **bir ekibin benzersiz stiller sağlamak yerine component oluşturmak için classname’leri oluşturmayı nasıl benimseyebileceğini düşünüyorum.** Son olarak, tekniklerden birini başarılı bir şekilde kullanmak için gerçekleşmesi gereken build steps leri keşfediyorum.

Bu kitap için, **component’lerimize stil vermek için CSS kullanacağız**. Tartıştığımız diğer seçeneklerin tümü CSS bilgisine bağlıdır. CSS için öğrendiğimiz şeyler diğer libraries için neredeyse her zaman değerli olacaktır, ancak aynı şey diğer teknikler için geçerli olmayabilir.

* **Building a Theme**

Bir component library oluştururken, **componentlerimizin tutarlı(consistent) color, spacing, fonts vb. kullanmasını isteriz.** Hiç, yalnızca nasıl göründüğüne bağlı olarak, kod tabanının farklı bir alanından farklı bir ekibin sorumlu olduğunu söyleyebileceğiniz bir uygulama kullandınız mı?(Have you ever used an app where you could tell a different team was responsible for a different area of the codebase based solely(yalnızca,sadece) on how it looked?)Bu senaryodan kaçınmak istiyoruz ve bunu temaların kullanımıyla başaracağız.

**Global template** tanımlamak için uygulamamızda kullanacağımız özel özellikleri içeren bir CSS file oluşturarak başlayacağız. **Kod tabanımızdaki(codebase) herhangi bir element uygulanan değişkenleri tanımlamak için :root kullanacağız:**

:root {

--background: #fff;

--text: #222;

...

--border-default: 1px solid;

--font-sm: 12px;

--spacing-sm: 4px;

...

--spacing-xlarge: 48px;

--shadow-default: 10px 9px 33px -17px rgba(0, 0, 0, 0.75);

--radius-default: 8px;

}

**Next, we’ll define our card styles:**

.card {

background-color: var(--background);

color: var(--primary);

padding: var(--spacing-medium);

margin: var(--spacing-large);

border: var(--border-default);

border-color: var(--light);

border-radius: var(--radius-default);

box-shadow: var(--shadow-default);

}

**Finally, we’re ready to use this style for our Card component:**

import React from "react";

import "../../../theme.css";

import "../../../Card.css";

export default function Card( ) {

return (

<div className="card">

... </div>

); }

Artık uygulamamızda componentleri nasıl şekillendireceğimiz konusunda oldukça iyi bir bilgimiz olduğuna göre, Rocket Coffee örneğimize geri dönelim ve bu bulguları orada uygulayalım. **theme.css dosyamızı /src/theme.css** dizinine kopyalayarak başlayacağız. Ardından, **/src/ component/atoms/Card/ altında Card.css** adında yeni bir dosya oluşturacağız:

* **Key Takeaways**

Bu bölümde, uygulamalarımızı şekillendirmek için kullanabileceğimiz çeşitli stratejileri ve çeşitli tekniklerin bazı **artılarını** ve **eksilerini** tartıştık. Ayrıca, our exploration için CSS kullanmanın neden sağlam bir seçim olduğunu tartıştık. Son olarak, **CSS custom properties(variables)** ve bunları uygulamamız için **temalar(themes)** oluşturmak üzere nasıl kullanabileceğimizden bahsettik.

https://developer.mozilla.org/en-US/docs/Web/CSS/Using\_CSS\_custom\_properties

**-Chapter 6 Ensuring the Quality of Our Components(72-82)**

Şimdiye kadar componentlerimizi izole bir şekilde oluşturduk ve onları kalite açısından Storybook aracılığıyla **manuel olarak test ettik**. Componentlerimizi doğrudan Storybook üzerinden çalıştırarak kodumuzun çalıştığını doğruladık, **ancak kaliteli bir kod tabanı sağladığımızdan emin olmak için uygulamak isteyebileceğimiz başka teknikler de var.** **Her sürümden(release’den) önce bir web uygulamasındaki her componenti kontrol etmeyi hayal edebiliyor musunuz?**

* **Unit Tests**

Storybook, bileşenlerimiz için mükemmel bir çalışma alanıdır. Şimdiye kadar, bileşenlerimizin beklendiği gibi çalışmasını sağlamanın birincil yöntemi olarak da hizmet etti. Bu, daha küçük örneklerimiz için harika çalışıyor, ancak biz profesyonel uygulamalar oluşturma işindeyiz. Bu profesyonel uygulamalarda genellikle oyunda çok sayıda bileşen bulunur. Her şeyi manuel olarak test etmek istediğimiz kadar iyi ölçeklenmeyecektir. Neyse ki, bu süreci **birim testi(unit test)** yoluyla otomatikleştirebiliriz.

**Birim testleri(unit tests)**, **belirli bir uygulama kodu birimini test etmek için kullanabileceğimiz kod bloklarıdır(block of code).** Başka bir deyişle, k**od tabanımızın çeşitli öğelerinin amaçlandığı gibi çalıştığından emin olmak için kod yazıyoruz.** Bu, kodumuzun çalıştığını doğrulamak için kod yazmak garip görünebilir, ancak testler oluşturmak, daha az hata ve daha yüksek değişim **esnekliği(resiliency)** ile zaman içinde daha iyi ölçeklenen kodla sonuçlanabilir. **Tüm bileşenlerimizi ve özelliklerimizi manuel olarak incelemek yerine, uygulamamızın istendiği(intended) gibi çalıştığını doğrulamak için bir test script çalıştırabiliriz.**

JavaScript has a built-in way that we can perform unit tests:

> console.assert(1 + 3 === 7);

Assertion failed

Birlikte geçirdiğimiz zamanlarda testler yazmak için **Jest'i** kullanacağız. **Jest, Facebook tarafından oluşturulan açık kaynaklı bir unit test library’dir.** **Jest**, testlerimizi yürüteceğimiz **araç(tool)** olarak hizmet eder ve **birim test(unit test)** kodumuzu n**asıl yapılandırmamız(structure**) gerektiği konusunda rehberlik sağlar. Kullanım kolaylığını esneklikle dengeler ve oldukça geniş bir geliştirici ekosistemine sahiptir. Bu büyük ekosistem genellikle daha fazla insanın **karşılaşıp(encountered)** sorunları çözmesi ve daha fazla **eklenti(plugin)** ile sonuçlanır.

Basit bir test yazacağız ve işlerin beklendiği gibi çalıştığından emin olacağız. Ancak işlerin beklendiği gibi çalışıp çalışmadığını kontrol etmeden önce, sonuçlanması gereken bir senaryonun doğru olduğundan **emin olmak iyi bir uygulama olabilir.**

**başarısız bir testte aslında bir test başarısızlığına neden olur. Çalışıyor gibi görünen, ancak birkaç hatayı yakalayamayan testler gördüm. Hataları giderirken testler incelendiğinde, testlerin her zaman geçtiği ortaya çıktı.** Testler yanlış bir güvenlik hissi verdi. Bu durumdan kaçınmaya yardımcı olmak için, testimizi geçmeden önce başarısız olduğundan emin olarak başlayacağız:

Burada jest kontrolünü baktık. add() fonksiyonun testini yaptı çalışıyor mu çalışmıyor mu diye? Daha sonra **This is a great way to test standard JavaScript methods and classes, but how would you approach testing a React component?**

* **Testing React Components**

React componentlerinin standart JavaScript yöntemleri veya sınıfları olduğu doğrudur, ancak oluşturma yüzeyi kavramına sahip oldukları için bu bileşenlerde daha fazla nüans vardır. **Web'de, React componentleri DOM'a (belge nesne modeli) dönüştürülür.** Testlerin de **handle rendering** için bir yola ihtiyacı vardır. React bileşen testlerimizde bir **rendering surface elde etmek(obtain)** için yapabileceğimiz birçok şey var. **React Testing Library'ye odaklanacağız.**

Bence **React Testing Library** inanılmaz bir test deneyimi sunuyor. Bu paradigmada, bileşenlerimizi render için React Testing Library'yi kullanacağız ve ardından **React Testing Library**'nin sağladığı **query methods** kullanarak componentlerimizle etkileşime geçeceğiz.

Rocket Coffee için ürün listemize bazı testler ekleyelim. CLI aracılığıyla projemize Test Kitaplığı eklemeye başlayacağız:

npm install —save-dev @testing-library/react

Test edilebilecek şeyler hakkında düşünerek başlayacağız. Şahsen, **genellikle branching code paths (bir bağlamda görünen ve başka bir bağlamda görünmeyen şeyler) ve etkileşim noktaları olan item’ları düşünürüm.** İşlerin beklediğimiz item’ları oluşturduğundan emin olmak için daha fazla test ekleyebiliriz, ancak bunu sizin için bir alıştırma olarak bırakacağız.

Bu durumda, aşağıdakileri test etmek istiyoruz:

• Show (On Sale) when the component is on sale.

• Call onAddToCard when the button is enabled and clicked.

• Disable the button when sold out.

ProductListItem.js'nin yanında yeni bir ProductListItem.test.js dosyası oluşturacağız. Test dosyasını açacağız ve ilk testimizi oluşturacağız. Unutmayın, Jest için herhangi bir şey içe aktarmamız gerekmeyecek, ancak React, React Testing Library ve componentimiz için içe aktaracağız.

Bu bileşenleri test etmeye **başlamadan(embark**) önce hızlı bir şekilde tartışmak iyi olacak bir şey: **React Testing Library, bu örneklerde kullanmayacağımız birkaç yardımcı yöntem ve fonksiyona(helper methods and functions) sahiptir.** It would be good to familiarize yourself a bit with the **documentation** at some point, as these **helper functions** may save you a lot of time and **debugging heartache** if you’re aware of them.

note: Keep in mind it can be a good practice to ensure that the test fails so we don’t think that the test works when it does not. **Tests that provide false security are worse than no tests.**

Bu testi aradan çıkardığımıza göre, devam edelim. **isSoldOut prop mevcut olduğunda düğmeyi devre dışı bıraktığımızı** kontrol etmek için testle başlayacağız:

it('disables the button when disabled', () => {

const { getByText } = render(<ProductListItem

name="Mocha"

price={3.50}

isSoldOut

/>)

expect(getByText("Sold out"))

.toHaveAttribute('disabled');

});

Eklemek istediğimiz son test, **addToCart'a** tıklamanın gerçekten bir **olayı tetiklediğinden( fires an event)** emin olmak içindir. Bir **Jest mock function** kullanacağız. **A mock function**, **gerçek bir fonksiyon yerine test ederken kullanabileceğimiz bir şeydir.** **Mock function’la jest.fn** olarak etkileşime geçeceğiz. Aşağıdaki örnekte, bu **Mock function’a** **onAddToCart'ta** bir değer ayarlıyor ve onu **componentimize** referans alıyoruz.

**Mock function’a** ek olarak, **onAddToCart** yönteminin çağrıldığından emin olmak için testimizde düğmeyi tıklamak isteyeceğiz. **React Testing Library, hedeflerimize ulaşmak için kullanacağımız bir “fireEvent” yardımcısına sahiptir:**

Kod:

import { render, **fireEvent** } from '@testing-library/react';

...

it('calls callback when button Add to Cart pressed', () => {

const onAddToCart = jest.fn();

const { getByText } = render(<ProductListItem

name="Mocha"

price={3.50}

onAddToCart={onAddToCart}

/>)

fireEvent.click(getByText("Add to Cart"))

expect(onAddToCart).toHaveBeenCalled()

})

* **Testing AlongSide StoryBook**

**Bu testler biraz tanıdık gelebilir. Bunun nedeni, StoryBook storylerimize çok benzemeleridir. Hem Storybook story’lerinde hem de testlerde, componentlerimizi farklı durumlarda inceliyoruz. Örneğin, standart bir state(durum), “On Sale” state ve "Sold Out" state’imiz var.**

What if we could eliminate the areas where we’re defining the JSX elements to render and leverage what we already set up for Storybook instead? **We can, and there are a couple of immediate(doğrudan) advantages to this:**

1) **Testlerimizde görsel olarak (“debug”)“hata ayıklayabiliriz”**. Testlerimizden biri başarısız olursa, **instead of logging out the rendered HTML (a common debugging technique), yerine Storybook'u başlatabilir** ve aynı adımları manuel olarak uygulayabiliriz. Test ve story, bir elementin nasıl oluşturulacağını belirlemek için aynı kodu kullandığından, kodumuzda manuel olarak gezinme ile otomatik test arasında uyumsuzluk yaşamadığımızdan emin olabiliriz.

2) **Storylerimizi güncel tutuyoruz.** Storybook'un sonunda bir ürünün mevcut durumuyla uyumsuz hale geldiği birçok proje gördüm. Yararlı bir araç olarak başlayan şey, geliştiriciler hikaye oluşturma ve sürdürme konusunda **daha az ilgili(relevance)** hale geldikçe alaka düzeyini kaybetti. **Bu bir trajedi çünkü Storybook çok fazla değer sağlayabilir, ancak Storybook'u test sürecinin bir parçası olarak ele almak, Storybook'un güncel kalmasını sağlamamıza yardımcı olur.**

3) **Daha az kod/daha az duplication.** İşler(things) aynı olmadığında duplication daha iyi bir seçenek olsa da, componentlerimizi Storybook'ta farklı durumlarda çalıştırmak istiyoruz. Benzer şekilde, **birim testlerimizde(unit tests) componentlerimizi çeşitli durumlarda test etmek istiyoruz. ProductListItem** örneğimizi alın. **Ürün stoktayken componentin bir state’e ve ürün tükendiğinde(sold out) component’in başka bir state’ine sahibiz.**

* **Key Takeaways (Önemli Çıkarımlar)**

**Bu bölümde, componentlerimizi otomatik olarak test etmenin yollarını ve bunu neden yapmak isteyebileceğimizi tartıştık. Jest ve React Testing Library'yi üst düzeyde tartıştık. Son olarak, unit testlerimizi güçlendirmek için mevcut Storybook hikayelerimizden nasıl yararlanabileceğimizi konuştuk.**

**-Chapter 7 Interacting(Etkileşim) with API Data (83-93)**

Şimdiye kadar, uygulamamızın küçük bir kısmı için **very** **specific componentler üzerinde çalıştık.** Bu bölümde, **componentlerimizin** nerede **yaşadığını** tanımlamanın ötesinde **tutarlılığı(consistency) promote** etmenin yollarını tartışacağız.

Bölüm 4'te, **classifying our component types ve bileşen sınıflandırmasının**, bu dosyaların kod tabanımızda nerede yaşadığı üzerinde nasıl bir etkisi olması gerektiğinden bahsettik. Ayrıca **atoms, patterns ve screen veya page componentlerini kullanmay**ı tartıştık. **Atoms, sistemimizdeki en düşük seviyeli yapı taşı componentleriyken, patterns biraz daha karmaşık yeniden kullanılabilir componentlerdir.** Son olarak **screens** bahsettik**; bunlar sayfa düzeyindeki(page-level) özellikler olacaktır.**

Ne yazık ki, çok büyük sistemlerde, **patterns** ve **screens** dediğimiz şeyler arasında çok fazla boşluk olabilir. Yani, şimdiye kadar çoğunlukla **componentleri reusable** bir şekilde oluşturmaya odaklandık ve **her şeyi uyumlu bir uygulamada nasıl birleştirmemiz gerektiğine odaklanmadık.** Yazılım geliştirmedeki herhangi bir şeyle, emrimizde epeyce seçenek var. Bu **seçeneklerden birkaçını ve çeşitli teknikleri kullanırken göz önünde bulundurmamız gereke**n bazı **trade-off’ları** tartışacağız.

* **Some Main Considerations (Bazı Önemli Hususlar)**

Göz önünde bulundurmamız gereken önemli hususlardan bazıları, **data loading(veri yüklemeyi)**, uygulama **state management(durumu yönetimini)** ve **routing(sayfalar arasında yönlendirmeyi)** nasıl ele aldığımızdır.

1) **Data Loading** – **Kodun bu kısmı, front-end uygulamamızın dışındaki sistemlerle** nasıl **etkileşim(interact) kurduğumuzu ifade eder.** Bu, **API servers veya third-party(üçüncü taraf) platformları gibi bir şey olabilir.**

2) **State Management** – Uygulamaları oluştururken göz önünde bulundurmamız gereken en zor şeylerden biri, u**ygulama state’ini nasıl yönettiğimizd**ir. Birçok senaryoda, **pages ve components** arasında paylaşılan **variables(değişkenleri)** takip etmemiz gerekecek. Ek olarak, çeşitli **componentler** **arasında uygulama state’in nasıl geçileceğini belirlememiz gerekir.** Tek başına bu konuyla ilgili bütün kitaplar olabilir, ancak bu kitapta React'in **built-in state management** kullanacağız.

Ek olarak, **uygulamamızı, klasör hiyerarşimizin state variable’larımızı nereye yerleştireceğimizi determine(belirlememize) yardımcı olabileceği şekilde yapılandıracağı**z. Bu bölümde ilerledikçe bunu daha fazla eylemde göreceğiz.

3)**Routing** – Çeşitli sayfalarımızı **birbirine bağlarız(linkleriz)** ve **yönlendirmeli tarayıcının URL'si(the browser’s URL with routing)** aracılığıyla uygulamamıza **giriş noktaları(entry points)** sağlarız**. Standart static HTML belgelerine** çok benzeyen front-end uygulamalarımız, tarayıcı çubuğundan(**browser bar**) yanıt verdikleri yollara sahiptir.

Sistemimizi y**önlendirme kavramı(concept of routing**) etrafında düzenleyeceğiz. Bunun anlamı şudur: **src/components/pages folder’ımız**, kahve dükkanımızın müşterilerinin kahve satın almak için ziyaret edeceği **rotalara(routes)** uyacak şekilde düzenlenecektir.

Bu, her **ana rota(main route)** için **pages** dizinimizde karşılık gelen bir klasöre sahip olacağımız ve **her alt yolun** o dizindeki **belirli bir alt componentle eşleneceği anlamına gelir.** Diyelim ki **uygulamamızda bir Checkout screen** olacak ve ödeme süreci, müşterinin **ödeme bilgilerini(payment information)** gireceği bir **ekran** ile **siparişini ve ödeme bilgilerinin(order and payment information)** doğru olduğunu **onayladığı(confirm)** bir **ekrandan** oluştuğunu varsayalım. Bu ikinci screen’de, satın alma işleminin gerçekten gerçekleşeceği yer burasıdır. Sayfalarımızdaki klasör yapımız biraz şöyle görünebilir:

RocketCoffee

|\_ src

|\_\_ pages

|\_\_\_\_ Checkout

|\_\_\_\_\_\_ Checkout.js

|\_\_\_\_\_\_ Confirm.js

|\_\_\_\_\_\_ index.js

|\_\_\_\_\_\_ PaymentInfo.js

Ekranlarımız daha sonra **/checkout**, **/checkout/paymentInfo** ve **checkout/confirm** ile eşlenir. Biraz sonra bunu ayarlayacağız, ama şimdilik, inşa ettiğimiz çeşitli parçaların nasıl birbirine bağlandığına dair zihinsel modelimizi oluşturuyoruz.

**Artık routing structure kuruldu ama uygulama state ve data loading nasıl halledeceğiz?**

* **Feature-Based Development**

**Data loading’imizin nerede gerçekleşeceğini belirlemek** için **feature-based development** adı verilen bir teknikten ipucu alacağız. **Feature-based development’**da kodumuzu **özellikler(features)** etrafında **düzenleriz(​​arrange**). **Features**, uygulamamızdaki bir **değer birimini(unit of value**) temsil eder ve özelliğin ihtiyaç duyabileceği her şeyi kapsar.

Bu paradigmayı benimsemeyen uygulamalar genellikle dosyaları **teknoloji etrafında düzenler(organize)**. Bu özellikle **Redux kullanan uygulamalarda belirgindir.** Uygulamamızda Redux kullanmıyor olsak da, bu örnek, oyundaki teknolojiler etrafında kod tabanımızı düzenlerken maruz kalabileceğimiz ek yükün bir kısmını aydınlatabilir.

**Redux** ile **shopping cart functionality** oluşturduğumuzu varsayalım. Geleneksel bir uygulamada, **componentler**, **pages** ve **Redux**, **actions**, **reducers**, **selectors** vb. ile kullanılan çeşitli teknolojiler için **klasörler(folders)** vardır. Böyle bir şeyle sonuçlanacaktık:

src

|\_ **actions**

|\_\_\_ cartActionCreators.js

|\_ **components**

|\_\_\_ cart

|\_\_\_\_\_ Cart.js

|\_ **reducers**

|\_\_\_ cartReducers.js

|\_ **selectors**

|\_\_\_ cartSelectors.js

**Product lists, user accounts,**ve diğer her şey teorik olarak **actions, reducers, selectors,** vb. arasında dağılmış dosyalara sahip olacaktır. Bu işe yarar ve bir miktar **tutarlılık(consistency)** sağlar, ancak dosyalar kod tabanı boyunca dağılmış olduğundan, bir şeyleri **güncellemek, taşımak veya silmek istediğinizde sorun olabilir.**

**Feature-based** bir codebase’de, **sepetimiz(cart)** için bir temel klasör elde ederiz ve **alışveriş sepetinin(cart)** tüm benzersiz öğeleri, teknoloji tarafından **spread out** yerine bu klasörde yaşar:

src

|\_ features/

|\_\_\_ cart/

|\_\_\_\_\_ **components**/

|\_\_\_\_\_ **actionCreators**.js

|\_\_\_\_\_ **index.js**

|\_\_\_\_\_ **reducer**.js

|\_\_\_\_\_ **selectors**.js

**Feature folders** doğrudan kullanmayacak olsak da, bir düzeyde, **pages folder’larımıza** biraz **feature** gibi davranacağız. Yani, **sayfaya özgü her şey sayfalar klasöründe** yaşayacaktır. Bunun anlamı, **data loading ve state management mekanizmalarımızı temsil ettikleri sayfanın yakınında tutacağız.**

Oluşturulan bu zihniyetle, uygulamamızın **bu 7. bölümünde** kullanabileceğimiz **data loading ile product list page’imizi** güncelleyeceğiz. **8. chapter’da, routing ve Storybook dışında sayfa ekranlarımızı nasıl görüntüleyebileceğimize bakacağız.**

* **Loading Data**

**Data loading** için, doğrudan **Products componentin de** bir **API request** gerçekleştirerek başlayacağız. Storybook hikayemiz **Products.stories.js** için kardeş dosyalar ve bir **index.js** dosyasıyla birlikte **src/screens/Products/Product.js** dosyasında **Products adlı bir component** oluşturacağız.

**Products componentimizde**, daha önce oluşturduğumuz **ProductList** kullanacağız ve bir **API'den** alınan verileri işleyeceğiz. İlk componentimiz biraz şöyle görünebilir:

**import React, { useEffect, useState } from 'react'**

**import ProductList, { statusTypes } from '../../components/patterns/ProductList'**

**export default function Products() {**

**const [productState, setProductState] = useState({**

**data: [],**

**status: statusTypes.loading**

**})**

**return <ProductList data={productState.data} status={productState.status} />]}**

Daha sonra Products.stories.js kısımlarını gösteriyor.

Sonra Products componentin içine fetch data kodunu gösterdi.

import React, { **useEffect, useState** } from 'react';

import ProductList, { statusTypes } from '../../components/ patterns/ProductList';

export default function **Products**() {

const [ productState, setProductState ] = useState({

data: [],

status: statusTypes.loading

});

**useEffect(() => {**

**const getData = async () => {**

**try {**

**const productFetch = await fetch('/api/products');**

**const productResponse = await productFetch.json();**

**setProductState({ data: productResponse.data, status:**

**statusTypes.loaded })**

**} catch (ex) {**

**console.error(ex);**

**setProductState({ data: [], status: statusTypes.errored })**

**} }**

**getData();**

**}, []);**

return <ProductList

data={productState.data}

status={productState.status}

/> }

**useEffect** is a little bit like **componentDidMount in class-based React components.**

Note: Normalde bir useEffect statement içinde başka bir **method kullanmamıza gerek olmadığını unutmayın. async/await'i** kullanmak istediğimizd**e ise useEffect doğrudan async olamadığı için başka bir yöntem oluşturuyoruz.**

Ancak Storybook'u çalıştırmak bir sorunu aydınlatabilir. Bir API ile etkileşime girmeye çalışıyoruz. Hikayelerimizde ve testlerimizde, verilerimizi elde etmek için gerçek bir API kullanmak istemiyoruz.

Bu, componenti bir error state’de görebilmemiz harika; ancak ürün listesini bu ekranın standart görünümü olarak görmeye çalışıyoruz.

**Screen/page-level components’miz** için Storybook'tan hal**a değer(value) elde edebildiğimizden emin olmak için kullanabileceğimiz birkaç strateji var.**

* **Container/Presentational Components**

**Deneyebileceğimiz ilk şey, componentlerimizi bir Container ve View componentlerine** bölmek. **Container componenti, verilerin alınmasından sorumlu olacak** ve **View componenti yalnızca verilerin görüntülenmesinden sorumlu olacaktır.** Bu stratejiyle, iki ayrı dosyaya sahip olurduk. **Products.container.js** ve **Products.js is the View component.**

Bu, **birçok projede başarıyla kullandığım bir strateji ama bir dezavantajı var. İhtiyaç duyabileceğinizden daha fazla dosya oluşturuyorsunuz. Daha fazla kod, hatalar ve korumanız gereken şeyler için daha fazla yüzey alanı olduğu anlamına gelir.**

* **Mock Data**

The technique we’re going to use in this book is **mocking the API endpoint for Storybook** (ve testlerimizi güçlendirmek için Storybook storylerini kullandığımızdan, testlerimizde de). **Bu functionality, açık kaynak aracı Mirage JS (miragejs.com) aracılığıyla sağlanır.** **Mirage, Storybook'ta component de görüntülerken API'ye yapılan tüm çağrıları verilerle değiştirirken** componentlerimizde **endpoint** ile etkileşime girmemize izin verecek. Miragejs'leri npm ile kuracağız:

npm install --save-dev miragejs

Once npm has finished, we’ll **update our test to build a mock API endpoint that’s used in Storybook**.

**Products.stories.js**

import { Server } from 'miragejs';

let server = new Server();

server.get('/api/products/', { data:

[

{ id: 1, name: "Mocha", price: 3.5, imageUrl:"https://source.unsplash.com/tNALoIZhqVM/200x100/",

},

... ]

});

Sunucuyu Miragejs'den içe aktarıyoruz ve **“/api/products/”** **isteğinde bulunduğumda bu verilerle yanıt ver” diyen bir mock kuruyoruz**. Şimdi Storybook'u çalıştırırsak, daha önce gördüğümüz hata yerine sahte verilerimizin döndürüldüğünü görebiliriz.

* **Wrapping Up**

**Bu bölümde, Screen/Page bileşenlerimize nasıl data load yapacağımızı tartıştık. Veri yüklemek için kullanabileceğimiz çeşitli tekniklerden ve Storybook stories ve unit testleri için verileri nasıl hızlı bir şekilde taklit edebileceğimizden bahsettik.**

**-Chapter 8 Building Our Application (94-106)**

**Component workspace**’imizin güvenliği için çalışıyoruz. Ne yazık ki, **product bir component library** olmadıkça, users, onları gerçek uygulamamıza yerleştirmedikçe componentlerimizden fazla bir değer almayacaklardır. **Bu bölümde, Storybook kullanarak şimdiye kadar yaptığımız her şeyi alıp gerçek bir uygulamaya uygulayacağız.** **Storybook'a yaklaşımımız**, **inşa edilen(built) uygulama sayısından biraz farklıdır,** ancak bu bölümde göreceğimiz gibi, gerçek uygulamamızı oluşturmaya hazır olduğumuzda, her şey oldukça hızlı bir şekilde yerine oturur.

* **Navigating Between Pages (Sayfalar Arasında Gezinme)**

**Uygulamamızda ihtiyaç duyacağımız şeylerden biri, coffee alışverişçilerimizin(shoppers) farklı sayfaları ziyaret etmesinin bir yoludur.** Uygulamamızın hangi bölümünün aktif olduğuna bağlı olarak componentlerimizi basitçe gösterebilir ve gizleyebiliriz. Bu, küçük uygulamalar için işe yarayabilir, ancak ölçekte sürdürülemez olacaktır. Bu stratejiyle ilgili basit bir örnekle daha belirgin olabilecek başka sorunlar da var.

Bir **HomePage** ve **About** sayfasını temsil eden bazı temel sayfa bileşenlerimiz olduğunu varsayalım:

const Home = () => <h1>Hi from the home page!</h1>;

const About = () => <h1>Hi from about!</h1>;

Ardından, **navigation itemlardan** birine tıklandığında çağrılan bir **callback** alan temel bir **Navigation componenti** ekleyeceğiz. Bu durumda, şu anda teknik olarak tipik **link functionality** yerine **button**-like **functionality** sağladığımız için bir **button** kullanacağız (**yani, tarayıcı yolunu değiştirmiyoruz ve bunun yerine hangi içeriğin mevcut olduğunu değiştiriyoruz** - **bazı nüans burada, ancak bunu daha sonra düzelteceğiz):**

export const **Navigation** = ({ onChangeNavigation }) => {

return (<nav>

<ul>

<li>

<button onClick={() => onChangeNavigation("home")}> Home</button>

</li>

<li>

<button onClick={() => onChangeNavigation("about")}> About</button>

</li>

</ul>

</nav>); };

Ardından, Application componentimizi aşağıdaki gibi ekleyeceğiz. Bu **App componenti**, sayfalarımızı ve navigasyonumuzu kullanacaktır:

function **App**() {

const [currentPage, setCurrentPage] = useState("home");

let content = undefined;

if (currentPage === "home") {

content = <Home />;

}

if (currentPage === "about") {

content = <About />;

}

return (

<div className="App">

<Navigation onChangeNavigation={setCurrentPage} />

{content}

</div>

); }

export default App;

Önceki **App bileşeninde**, **sayfa düzeyinde içeriğimizi tutacak ve duruma göre içeriği değiştirecek bir değişken, içerik tanımlıyoruz.** Bu **yaklaşımda gördüğünüz bazı sorunlar nelerdir?**

Bu oldukça **basit(straightforward)** görünse de, bu stratejiyi kullanırken karşılaşabileceğimiz bazı kesin zorluklar var. Her şeyden önce, **bu iyi ölçeklenmeyecek(scale).** **İki sayfa için neler olduğu oldukça açık, ancak 20 veya daha fazla sayfamız(page) olsaydı bunu nasıl hallederdik?** Geliştirici deneyiminin ötesinde, bu teknik, **coffee shoppers’imize de harika bir deneyim sağlamayacaktır.**

Bir kullanıcı **About sayfasın**a **bookmark koymak** isterse ne olur? Bookmark’ını her ziyaret etmeye çalıştıklarında, bir **URL parameters dayalı olarak etkin sayfayı yeniden sulandırmak(rehydrating)** için bir mekanizma **oluşturmadıkça, bu App state tarafından kontrol edildiğinden ana sayfaya(main page) geri dönerlerdi.**

Bu noktada kendimize **“Router kavramını yeniden mi icat ediyoruz?**” diye soruyor olabiliriz. Bu sorunun cevabı "**Evet**" olacaktır. **Neyse ki, kullanabileceğimiz mükemmel bir router var.**

* **Routing**

Müthiş **React Router** kitaplığını kullanacağız. Şimdi örnek uygulamamızı aşağıdaki şekilde değiştirebiliriz:

export default function App() {

return (

<BrowserRouter>

<Navigation />

<Switch>

<Route path="/about" component={About} />

<Route path="/" component={Home} />

</Switch>

</BrowserRouter>

);

}

Oradan, daha önce kullandığımız **button öğeleri yerine React Router tarafından sağlanan Link componentini** kullanmak için Navigasyon componentimizi güncelleyeceğiz:

export const **Navigation** = () => {

return (<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/about">About</Link>

</li>

</ul>

</nav>); };

Bu strateji ile **BrowserRouter** tanımlıyoruz; bu, **routerımız** için **top-level provider’dır** ve bir **Switch component** içinde yer alan rotalarımızı oluşturur. **Switch bileşeni temel olarak aynı anda sadece bir rotanın aktif olduğunu söylememizi sağlar.** Kavramsal olarak, bunu, **sağlanan değerlere dayalı olarak yalnızca bir kod yolunun yürütüleceği bir switch ifadesi gibi düşünebiliriz.** Route bileşenleri bir **path(yol)** ve verilen **yola dayalı olarak etkin olması gereken componenti** alır.

**Navigation componetimiz** için, React Router'dan Link componentindeki buttonları güncelledik. **Link bir anchor tag’ı(<a/>) olarak işlenir** ve adından da anlaşılacağı gibi başka bir rotaya bağlanır. **To prop** **(yani, <Link to=“/about”>)** aracılığıyla Link bileşenlerimizin **route path’ini** sağlarız.

**Router kullanmanın**, duruma vb. dayalı olarak içerik görünürlüğünü **elle işlemeye göre sayısız avantajı vardır.** **İlk olarak, React Router'ı kullanmak, duruma göre içerik görünürlüğünü manuel olarak yönetmekten biraz daha iyi ölçeklenir(scale).** **Switch ifademize routes eklemeye devam edebilir, hatta iç içe(nested) routes** sağlayabiliriz (iç içe rotalar hakkında **daha fazla bilgi için React Router docs** bakın).

React Router, URL rota güncellemeleri sağlar. Önceki yönlendirici örneğini çalıştırırsanız, bir bağlantıya tıkladığınızda URL'nin güncellendiğini fark edebilirsiniz. Bu, kullanıcıların bulundukları sayfalara **bookmark** koyabilecekleri anlamına gelir. Ek olarak, durumu değiştiren buttons yerine gerçek bir **anchor tags** olduğu için **Link componenti** daha **mantıklıdır(sensible).** **Buttons, callback için bir öğe olarak işe yaradı, ancak aslında router ile U**RL'yi değiştirdiğimiz için bir **anchor tag’ı** daha iyidir.

* **Updating Our Application to Use Routes**

Rocket Coffee uygulamamızı rotaları kullanacak şekilde güncelleyelim. Bölüm 7'den beri, **Cart** ve "**My Account**" için bazı sayfalar ekledim. Gerçek bir veritabanına veya herhangi bir şeye bağlı değiller, ancak router ile uğraşmak için yeterli olmalıdırlar.

Sistemimizi router kavramı etrafında düzenlemek istiyoruz. **Bunun anlamı, src/component/pages folder, kahve dükkanımızın müşterilerinin kahve satın almak için ziyaret edeceği routers eşleşecek şekilde taşınacağıdır.**

Bu, her **main router** için pages dizinimizde **karşılık gelen(corresponding)** bir folder’a sahip olacağımız ve her **subroute** o dizindeki belirli bir **subcomponent** eşleneceği anlamına gelir. **File structure** aşağıdaki gibi görünebilir:

RocketCoffee

|\_ src/

|\_\_ components/

|\_\_\_\_ atoms/

|\_\_\_\_ patterns/

|\_\_ screens/

|\_\_\_\_ Cart/

|\_\_\_\_ Products/

|\_\_\_\_ UserProfile/

**Screens klasörlerimizin** her biri bir component, bir **index file** ve bir **story içerir** (ve isteğe bağlı olarak bir test ve stil bilgisi içerir). Storybook üzerinden sayfalarımızı görüntüleyebiliyoruz ama gerçek anlamda işleyen bir uygulama istiyoruz. Devam edelim ve **React Router'ı ekleyelim ve gerçek uygulamamızı Storybook dışında kullanalım.**

Npm'den react-router-dom paketini ekleyerek başlayacağız. Bir web uygulaması geliştiriyoruz, bu nedenle istediğimiz React Router'ın DOM versiyonu. Doğrudan tarayıcıyla etkileşime giren bazı alanları vardır:

npm install react-router-dom

export default function Routes() {

return (

<BrowserRouter>

<Switch>

<Route path="/cart" component={Cart} />

<Route path="/userProfile" component={UserProfile} />

<Route path="/" component={Products} />

</Switch>

</BrowserRouter>

); }

**Mock.server.js içine;**

import { Server } from "miragejs";

const productData = [

{

id: 1,

name: "Mocha",

price: 3.5,

imageUrl: "https://source.unsplash.com/tNALoIZhqVM/200x100/",

}, {

id: 2,

name: "Latte",

price: 3.5,

imageUrl: "https://source.unsplash.com/tNALoIZhqVM/200x100/",

}, {

id: 3,

name: "Vanilla Latte",

price: 3.5,

imageUrl: "https://source.unsplash.com/tNALoIZhqVM/200x100/",

}, ];

const server = new Server({

routes() {

this.namespace = "/api";

this.get("/cart", () => ({ data: [productData[1]] }));

this.get("/products", () => ({ data: productData }));

this.get("/profile", () => ({

name: "Saige McDermott",

memberSince: "June 12th, 2018",

recentOrders: [

102

{ orderId: 12, name: "Latte" },

{ orderId: 27, name: "Cafe Miel" },

],

}));

this.get("/users", () => [

{ id: "1", name: "Luke" },

{ id: "2", name: "Leia" },

{ id: "3", name: "Anakin" },

]); },

});

export default server;

**App.js içine;**

import React from "react";

import './Mock.server'

import Routes from './Routes';

function App() {

return (

<Routes /> );

}

export default App;

* **Navigation**

**Navigation.js:**

// patterns/Navigation/Navigation.js

import React from "react";

import { Link } from "react-router-dom";

export default function Navigation() {

return (

<nav>

<ul> <li>

<Link to="/">Products</Link>

</li>

<li>

<Link to="/cart">Cart</Link>

</li> <li>

<Link to="/userProfile">Profile</Link>

</li>

</ul> </nav>

); }

**We’ll import these screens in our *Routes.js* file and place them in a React Router browser router as follows:**

import React from "react";

...

import Navigation from "./components/patterns/Navigation/ Navigation";

export default function **Routes**() {

return (

<BrowserRouter>

**<Navigation />**

<Switch>

<Route path="/cart" component={Cart} />

<Route path="/userProfile" component={UserProfile} />

<Route path="/" component={Products} />

</Switch>

</BrowserRouter>

); }

**-Chapter 9 Automating Repetitive Tasks(Tekrarlayan Görevleri Otomatikleştirme) (107-116)**

Bu bölümde, geliştirmemizi nasıl **hızlandırabileceğimizi** inceleyeceğiz. Uygulamamızı bir dizi daha küçük parçadan oluşturmayı tartıştık. **Her page componenti**, **pattern component** ve **atom component** için bir component, bir **index file, bir test file, bir story ve çoğu durumda bir style file oluşturuyoruz. Sizi bilmem ama bu benim yapmamayı tercih ettiğim çok fazla çaba. Söylemesi garip çünkü karmaşık bir iş değil ama sıkıcı.**

İnsanları bilgisayarların daha iyi yapabileceği işler için kullanmayın - bu gezegendeki tek gerçek kaynak olan insan enerjisinin ve yaratıcılığının israfıdır ve insan ruhunu küçük düşürür.(J. Paul Morrison, programmer and inventor )

Bu alıntıyı gerçekten takdir ediyorum. Bana göre, dünyayı kurtarmak gibi daha önemli şeylere odaklanabilmesi için işini daha iyi yapması için bilgisayarların yardım ettiği Tony Stark'ın (Demir Adam'ın ikinci kişiliği) düşüncelerini gündeme getiriyor. **Sizi bilmem ama keşke bir robot asistanım olsaydı.** **Bir projedeki daha önemli şeylere dikkatimi verebilmek için sıkıcı işleri bir bilgisayara devretmek isterim.**

Bilim kurgu hikayeleri kadar karmaşık bilgi işlem güçlerini kullanmayacak olsak da, işlerimizi kolaylaştırmak için bilgisayarlardan **yararlanabiliriz(leverage)**. **Directory structure oluşturmak(Constructing)** zor değil. **Bu işi daha basit(straightforward) hale getirmek için bazı yazılımları kullanalım.**

* **Our Own CLI**

Projemize özel bir CLI oluşturacağız. **CLI, bir komut satırı arabirimidir(command-line interface)**, yani komut satırında çalışan bir uygulamadır. **Create React App, npm, yarn veya git kullandıysanız, bir CLI ile etkileşime girmişsinizdir.** Dosyalarımızı ve dizinlerimizi istediğimiz yapıda **oluşturma( construct) i**şlemini neredeyse hiç düşünmeyi gerektirmeyen bir işlem yapacağız.

**Gluegun**, **Plop** ve diğerleri gibi bu amaca ulaşmak için kullanabileceğimiz **bir dizi tool** var. Ancak bu kitapta **Hygen (www.hygen.io/) kullanacağız.**

**Hygen, dokümantasyonu ile “Size zaman kazandıran ölçeklenebilir kod üreticisi” olarak kabul edilir.** Birkaç başka oluşturucu/CLI aracı kullandım ve **Hygen'in net, sonuç-odaklı mimarisini(results-driven architecture) takdir ediyorum. Belirli bir konumda Hello world fonksiyonuna sahip bir dosya oluşturacağımız basit bir örneğe bakalım**. Merak etme. Merhaba dünya heyecan verici değil, ancak yakında bu kavramları **Rocket Coffee kod tabanımıza uygulayacağız.** İdeal olarak, uygulamamızda **bir component oluşturmak için komut satırından Liste 9-1'i çalıştırabileceğiz.**

**Listing 9-1. Theoretical CLI code generator**

> yarn generate component UserForm —location atom

added: src/components/atoms/Button/index.js

added: src/components/atoms/Button/Button.js

added: src/components/atoms/Button/Button.stories.js

added: src/components/atoms/Button/Button.test.js

added: src/components/atoms/Button/Button.css

* **A Brief Example(Kısa bir Örnek)**

**Bilgisayarımıza Hygen ekleyerek başlayacağız.** Bununla başa çıkmanın birkaç yolu var, ancak bunu npm aracılığıyla global bir modül olarak eklemek iyi sonuç veriyor:

**npm install -g hygen**

**Note:** Hijyen aracını bilgisayarımıza eklemek istemiyorsak, benzer bir sonuç elde etmek için **npx'den** faydalanabiliriz. **npx aracı, npm kayıt defterinde barındırılan bir CLi aracını çalıştırabilmemizin bir yoludur.** npx'i kullanmayı tercih ediyorsanız, bu kılavuzda aracın küresel olarak kurulu sürümüne atıfta bulunduğumuz herhangi bir yerde, bunun yerine npx'i kullanın.

Örneğimiz için, bu kitap **boyunca(throughout)** inşa ettiğimiz Rocket Coffee projesini kullanacağız. Bununla birlikte, aynı adımların herhangi bir projede çalışması gerektiğini söyledi.

**Hygen, CLI komutları oluşturmak için template files ve surrounding(çevresindeki) folders kullanır.** “Merhaba dünya” örneğimiz için, **yeni HelloWorld adlı hygen fonksiyonunu (*hygen function new HelloWorld*** ) çalıştırmak istiyoruz. **Generator’ımızı** başlatmak için yapmamız gereken Hygen'i kurmanın ötesinde bazı şeyler var. Aşağıdakileri çalıştırarak başlayacağız:

**> hygen init self**

added: \_templates/generator/help/index.ejs.t

added: \_templates/generator/with-prompt/hello.ejs.t

added: \_templates/generator/with-prompt/prompt.ejs.t

added: \_templates/generator/new/hello.ejs.t

Bu ifade, Hygen'i diğer generator oluşturmak için kullanacağımız bir generators başlatır. Function generator oluşturmaya hazırız:

**> hygen generator new function**

**Hygen** projemizde **yeni bir klasör oluşturacak**. **\_templates/function/new klasörümüze gitmeli ve “hello.ejs.t” dosyasını düzenlemeliyiz.** **Hygen, editörlerin bu dosyaya ek IDE özellikleri uygulamaya çalışmasını önlemek için .t dosya türünü kullanır.(hello.ejs.t gibi)** IDE'nin bu template’leri bilmemesi muhtemeldir. Birazdan çalıştıracağımız **generator komutu direkt olarak klasör yapısına karşılık geliyor.** Örneğin, **hygen function new {Name} çalıştıracağız**, **burada Name, generator’a sağlayacağımız input’dur.** You may notice we have a folder for ***function*, the first statement in our command, and a function for *new* as well.**

**Note For this example, keeping the template name as hello should be fine. the template name is more important when we plan on keeping these files around; the name helps our teammates and “future us” recognize the intent of the template.**

**\_templates/function/new/hello.ejs.t** içindeki kodu aşağıdaki şekilde güncelleyeceğiz:

**---**

to: src/components/<%=**name**%>.js

**---**

console.log("hello world!")

**Hygen templateleri**, Node uygulamaları için popüler bir **template engine olan ejs'yi temel alır.** Buna aşina değilseniz endişelenmeyin, çünkü bir jeneratör inşa etmek için ejs'nin içini ve dışını bilmek gerekli değildir. **Amaçlarımız için, <%= %> arasındaki kodun dosyaya gönderileceğini anlamak istiyoruz.** Örneğin, belgemizin filename’ini doldurmak için variable **name’ini** kullanıyoruz.

ejs'ye ek olarak, **bu template**’in **iki bölümden oluştuğunu** anlamak önemlidir. Template’in “---” ile çevrili üst kısmı, şablonun **ön madde bölümüdür.** Şablon ayarları belgenin bu bölümüne aittir. “Merhaba dünya” örneğimiz için ön konuda, bu üretecin üretilen çıktısının **to özelliği olarak nereye ait olduğunu belirtiyoruz**. Bu oluşturucu için, **src/components** içine yerleştirilmiş belirli bir isime sahip bir dosya istediğimizi belirtiyoruz. Şimdi jeneratörümüzü çalıştırırsak ne olur?

Herhangi bir hatayla karşılaşırsanız, lütfen önceki adımları, **hygen init self** ve **hygiene generator new function** uyguladığınızdan emin olun. Hygen, şu adımlar olmadan beklediğimiz gibi çalışmayacaktır:

> **hygen function new Test**

Loaded templates: \_templates

added: src/components/Test.js (Test.js dosyasını src/components içine oluşturdu)

* **Building Some Generators for Our Project**

Elimizdeki bu bilgilerle, kod tabanımızda çalışmasını kolaylaştırmak istediğimiz oluşturucuları oluşturalım. Önceki metinde kullanılan işlev klasörünü \_templates/function iken \_templates/component olarak yeniden adlandırarak başlayacağız. Klasörü kaldırabilir ve oluşturucu işlemini yeniden başlatabilirsiniz, ancak ben genellikle bu yolu tercih ederim.

Oradan, \_templates/component/new/ konumuna gideceğiz, hello.ejs.t dosyasını **component.ejs.t** olarak yeniden adlandıracağız ve temel bir **index.ejs.t, component.stories.ejs** oluşturmak için bu dosyayı ve **component.test.ejs.t** dosyası kopyalayacağız. Bu şablonların her birine adıyla ilgili bir amaç vereceğiz. Component dosyasını düzenleyerek başlayacağız. Listing 9-2'deki kodu içermesini istiyoruz.

**Listing 9-2. component.ejs.t**

---

to: src/components/atoms/<%=name%>/<%=name%>.js

---

import React from 'react'

export default function <%= name %> () {

return <h1>Hello from <%= name %></h1>;

}

**Now, when we run our generator, we see that we create an *atom* component in our src folder:**

> hygen component new First

Loaded templates: \_templates

added: src/components/atoms/First/First.js

**index.ejs.t içindeki kod;**

---

to: src/components/atoms/<%=name%>/index.js

---

export { default } from './<%=name%>';

**component.stories.ejs.t içindeki kod;**

---

to: src/components/atoms/<%=name%>/<%=name%>.stories.js

---

import React from 'react'

import <%=name%> from './<%= name %>'

export default { title: 'atoms/<%= name %>', component: <%= name %>}

const Template = args => <<%= name %> {...args} />

export const Standard = Template.bind({})

Standard.args = {}

**component.test.ejs.t içindeki kod;**

---

to: src/components/atoms/<%=name%>/<%=name%>.test.js

---

import React from 'react'

import { screen, render } from '@testing-library/react';

import <%= name %> from './<%= name %>';

describe('<%= name %>', () => {

it('renders as expected', () => {

const { container } = render(<<%= name %> />);

expect(container).toMatchSnapshot();

}) })

Bunları ekledikten sonra terminale -> **hygen component new Deneme**

Loaded templates: \_templates

added: src/components/atoms/Deneme/Deneme.js

added: src/components/atoms/Deneme/Deneme.stories.js

added: src/components/atoms/Deneme/Deneme.test.js

added: src/components/atoms/Deneme/index.js

Testlerimizi veya Storybook'umuzu çalıştırabiliriz ve her şey beklendiği gibi çalışır ve yeni testlerimiz geçer. Bu harika ve çok zaman kazandırabilir; Ancak bir sorunumuz var. **Componetlerimizi atoms, patterns ve pages(veya screens)**  olarak sınıflandırdığımızı hatırlıyor musunuz? **Ne yazık ki, şu anda ürettiğimiz componentleri atom** olacak şekilde kodluyoruz**. Bunu düzeltelim.**

* **Adding Additional Variables to Our Generator**

**Generator’e** ek bir parametre olan **type'ı** kabul etmemiz gerekiyor. Bunu halletmemizin birkaç yolu olsa da (şablon(template) dosyalarımızdaki mantık gibi), yeniden kullanılabilir bir **helper** oluşturacağız. Tüm şablonlarımızın ne tür bir componentin oluşturulduğunu bilmesi gerekir, **bu nedenle bu reusable teknik, kodun tekrarlanmasını önlememize yardımcı olacaktır.**

**Hygen, varsayılan** Hygen yardımcılarına sahip top-level bir **h nesnesine** sahiptir. Projemizin root’unda **.hygen.js adlı bir dosya** oluşturarak ve şablonlarımızda kullanmak istediğimiz **fonksiyonlarla** bir **helpers object** oluşturarak bu nesneye eklemeler yapabiliriz (Listing 9-3). Komut satırından sağlanan türü alan **iki yardımcı işlev yapacağız: biri folder path**, diğeri **name of the type** almak için. Her olası type için değerleri tutacak bir nesne yaratacağız. Bu şekilde, bir değerin bulunduğu her helper function güncellemeye gerek kalmadan buna daha sonra eklemeler yapabilir veya bazı şeyleri değiştirebiliriz.

**Listing 9-3. .hygen.js in the root of our project**

const types = {

atom: { name: 'atoms', path: 'components/atoms/'},

pattern: { name: 'patterns', path: 'components/patterns/'},

screen: { name: 'screens', path: 'screens/'}

}

module.exports = {

helpers: {

getTypePath: (type = 'atom') => {

return types[type].path;

},

getType: (type = 'atom') => {

return types[type].name

}

} }

Şimdi bu **helper methodu** kullandığımızda, **generator’a** argüman olarak ilettiğimiz herhangi bir **type**, dosyaların nerede yaşadığını belirlemek için kullanılacaktır. Şimdi, yardımcı yöntemlerimizi kullanmak için **story şablonunu** güncelleyelim. Unutulmaması gereken bir şey, genellikle tür değişkenimize basitçe **type** olarak başvurabileceğimizdir; ancak bu, **type’ın** gerekli olduğu anlamına gelir. **Generator**, **type** belirtilmediğinde varsayılan olarak bir **atom** olacak şekilde ayarladık. **Type’ımızı elde etmek için doğrudan type değişkeni yerine Hygen'in locals nesnesini kullanacağız (Listing 9-4).**

***Listing 9-4.*** \_templates/component/new/component.stories.ejs.t

---

to: src/**<%= h.getTypePath(locals.type) %>**<%=name%>/<%=name%> .stories.js

---

import React from 'react'

import <%=name%> from './<%= name %>'

export default { title: '**<%= h.getType(locals.type) %>**/<%= name %>'}

export const Standard = () => <<%= name %> />

Bu yeni **helper methodu** kullanmak için **templates** geri kalanını güncelleyebiliriz. Bu tamamlandıktan sonra, **generator** çalıştırmak **atoms, patterns ve screen componetleri** için çalışır. Aşağıdaki komut, destekleyen **test, story ve index** dosyalarıyla **screens** dizinimiz altında yeni bir component oluşturacaktır:

**> hygen component new UserProfile --type screen**

Loaded templates: \_templates

added: src/screens/UserProfile/UserProfile.js

added: src/screens/UserProfile/UserProfile.stories.js

added: src/screens/UserProfile/UserProfile.test.js

added: src/screens/UserProfile/index.js

**-Chapter 10 Communicating Our Components (117-124)**

Şimdiye kadar çoğunlukla uygulamalarımızı geliştirmeyi daha kolay hale getirecek stratejilere odaklandık. Bununla birlikte, yazılım geliştirmede başarı, **yalnızca(merely)** iyi kod yazmaktan çok daha fazlasına bağlıdır. **Projeler, insanların ne kadar iyi iletişim kurduğuna bağlı olarak bir kazanç veya kayıp olabilir. Neyse ki(Thankfully) Storybook burada da bize yardımcı oluyor.**

Tahmin etmiş olabileceğiniz gibi Storybook, bir kod tabanında hangi componentlerin bulunduğunu diğer geliştiricilerle iletişim kurmamıza yardımcı olur. Bir geliştiricinin işlevselliğin başka bir yerde var olduğu hakkında hiçbir fikri olmadığı için aynı bileşenin veya kod parçacığının birçok yerde yaşadığı kaç projede bulundunuz? Tüm geliştiricilerin başvuruda bulunabileceği ve bileşenin çalışan bir sürümünü görebileceği bir bileşen çalışma alanı kullanmak bu iletişime yardımcı olur. Şimdiye kadar kullandığımız stratejiler, geliştiriciler için yararlıdır, ancak biraz daha fazlasını yapabiliriz.

* **Documenting Our Components**

Diğer geliştiricilerin **componentlerle** mümkün olan en kısa sürede hız kazanmasına yardımcı olmak istiyoruz. Yalnızca hikayelere sahip olmak bu **çabaya(endeavor)** yardımcı olur, ancak deneyimi, geliştiricileri **yalnızca(merely)** hikayelere güç veren koda bakmaya yönlendirmekten biraz daha güzel hale getirebiliriz. Hikayelerimize ek dokümantasyon özellikleri ekleyerek başlayacağız. Rocket Coffee kod tabanımızı kullanarak bu güncellemeleri yapmaya başlayalım.

Storybook bu **documentation** da bize yardımcı olabilir. Hikayelerimizin varsayılan dışa aktarımının bir nesne olduğunu hatırlıyor musunuz? Hikayelerimizle birlikte bazı belgeler sağlamak için kullanabileceğimiz bazı ek özelliklerimiz var.

**> yarn add @storybook/addon-docs**

Oradan, StoryBook **main.js** dosyasını içerecek şekilde güncellememiz gerekecek.

module.exports = {

stories: ["../src/\*\*/\*.stories.js"],

addons: [

"@storybook/preset-create-react-app",

"@storybook/addon-actions",

"@storybook/addon-links",

"@storybook/addon-knobs",

"@storybook/addon-docs" ]};

argTypes'ın bir parçası olarak sağladığımız bu açıklamalar özelliği, metnimize biraz hafif stil sağlamak için Markdown'ı destekler. Markdown'a aşina değilseniz, Mastering'i ziyaret etmenizi öneririm GitHub'dan Markdown'ı (<https://guides.github.com/features/>mastering-markdown/).

Varsayılan default export parameters.docs.description’a bir özellik ekleyeceğiz. hikayemizle birlikte göstermek istediğimiz açıklamayı içeren bileşen. **Bu alan aynı zamanda Markdown'ı da destekler.**

parameters: {

docs: {

description: {

component:

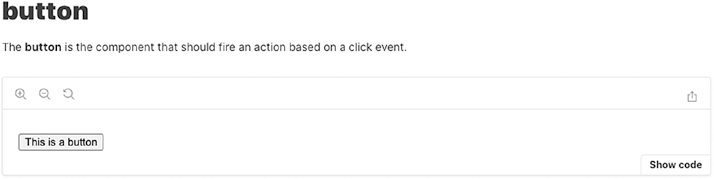
//\*\* button\*\* markdown’lı kısım

'The \*\*button\*\* is the component that should fire an action based on a click event.'

}

}

}

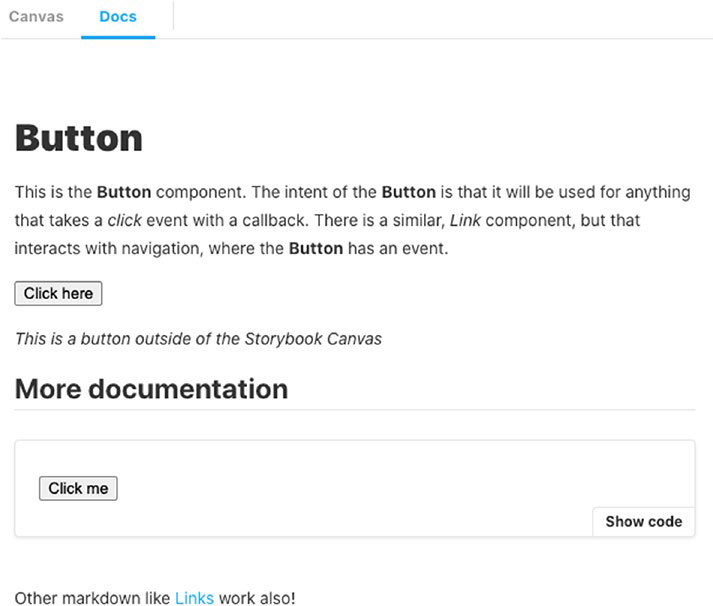


* **More Advanced Documentation**

Bu oldukça iyi çalışıyor, ancak belgelerimiz üzerinde daha fazla kontrol istiyorsak ne olacak? Storybook varsayılan nesnesiyle yapabileceğimiz pek çok şey var, ancak belgelerimizi tamamen bir JavaScript nesnesi aracılığıyla güncellemek sıkıcı olabilir. Neyse ki, belgeleri farklı bir strateji kullanarak sağlayabiliriz.

* **MDX**

Hikayelerimizi yazmak için **Markdown'u** veya daha spesifik olarak **MDX'i** de kullanabiliriz. Dokümantasyon sitesine (mdxjs.com) göre, "**MDX, JSX'i Markdown belgelerinize sorunsuz bir şekilde yazmanıza izin veren, yazılabilir bir formattır.**" Bu, StoryBook öykülerini belgelemenin isteğe bağlı bir yoludur, ancak amaç, belgelerin yanı sıra öyküler sağlamaksa tercih edilebilir.



Bu, bileşen çalışma alanımıza mükemmel belgeleme işlevselliği sağlar. Markdown'ın kolaylığı ve esnekliği, JSX'in gücüyle birleştiğinde bize dokümantasyon konusunda süper güçler veriyor.