

# Routing Kavramı

**Routing**, bir uygulamanın belirli bir URL'ye yanıt vermesini sağlayan yöntemdir.

- Kullanıcıların bir web sitesinde gezinmesini ve belirli sayfalara erişmesini sağlar, aradıklarını bulmalarını kolaylaştırır: **/profile , /ayarlar, /giris-yap gibi**
- Paylaşımı kolaylaştırır: amazonda bir ürün kategorisini **amazon.com/spor-ayakkabili/nike/erkek** olarak paylaşabiliriz, tarif etmekten çok daha kolay
- **SEO (Arama Motoru Optimizasyonu)** performansını iyileştirir, arama motorlarına hangi sayfaların hangi konularla ilgili olduğunu belirtir.
- **Browser history**: ileri/geri butonları ile yönetilebilir bir arşiv

# SSR: Server Side Rendering

- Görüntülenen sayfanın HTML kodu serverdan hazır gelir.
- Sadece görüntülenen sayfa kodu serverdan istediği için sayfa boyutları azdır;  
**ilk sayfa açılışı hızlıdır. (+)**
- Sayfalar içerikleriyle birlikte geldiği için **SEO performansı daha iyidir. (+)**
- Blog sayfaları, haber siteleri gibi kullanıcı **etkileşimi az sayfalar için idealdir.**
- Her sayfada ortak olan: sayfa yapısı, logo, footer, header gibi bölümleri tekrar tekrar serverdan ister ve sayfalararası geçişte tüm DOM baştan oluşturulur.  
**sayfalararası geçiş genelde yavaştır. (-)**

# CSR: Client Side Rendering

- Uygulamanın tüm sayfalarını oluşturacak HTML, CSS ve JS kodundan oluşan paket (**bundle**) serverdan gelir, sayfalar browserda oluşturulur.
- Tüm uygulama kodu tek seferde yüklenir, **ilk sayfa açılışı yavaştır (-)**
- Sayfalar içerikleriyle birlikte gelmediği için **SEO performansı genelde kötüdür. (-)**
- Web uygulamaları, sosyal medya siteleri gibi **çok etkileşimli sayfalar için idealdir.**
- Sayfalar arası geçişte yalnızca farklı bölümler değiştirilir, DOM en baştan oluşturulmaz. Bu sayede **sayfalar arası geçiş genelde hızlıdır. (+)**
- JS kütüphaneleri ve internet hız artışlarıyla birlikte daha çok kullanılır oldu.
- **Bir kez yüklenikten sonra hızlıdır (+), fakat normal sayfalara göre daha çok RAM tüketir. (-)**

# SSR

- + Ideal for static sites
- + Quick initial page access
- + No JS dependency
- + Easy-to-crawl site for better SEO
- Multiple server requests
- Full page reloads
- Non-rich site interactions
- Higher latency, prone to vulnerability

# CSR

- + Ideal for web apps
- + Fast site rendering after initial load
- + Rich site interactions
- + Robust JS library selection
- Negative SEO for incorrect rendering & API response delays
- Slow initial load time
- Requires external library
- Higher memory consumption

# React Router Kurulum

- **npm i react-router-dom@5**
- v5 Docs: <https://v5.reactrouter.com/web/guides/quick-start>
- Temel component'larını kullanırız: **BrowserRouter, Switch, Route**

# BrowserRouter, Switch, Route

## BrowserRouter (Provider Component)

- Genel routing wrapper (kapsayıcı)
- Kök (Root - App) componentini kapsamalı (**main.jsx, index.jsx**)

## Switch

- Route'ları kapsayan component (**ul gibi**)
- URL'e bakar, yalnızca eşleşen Route elemanın içindekini sayfada gösterir. (**if gibi**)

## Route

- URL yazılan path'e göre hangi component'in yükleneceğini belirtir.

# BrowserRouter, Switch, Route

```
import { BrowserRouter } from "react-router-dom";

const root = document.getElementById('root');
ReactDOM.createRoot(root).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

```
import { Switch, Route } from 'react-router-dom';

function App() {
  return (
    <div>
      <Switch>
        <Route path="/">
          <MainPage />
        </Route>
        <Route path="/geziler">
          <TravelBlogs />
        </Route>
        <Route path="/hakkimda">
          <AboutMe />
        </Route>
      </Switch>
    </div>
  );
}
```

# Link & NavLink

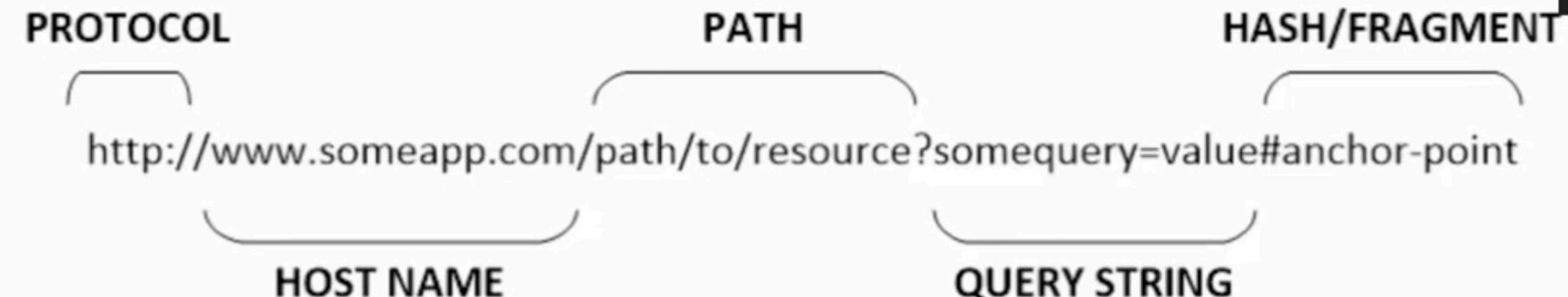
## Link

- Sayfalar arası geçiş için **a** tagları yerine kullanılır
- Aslında başka sayfaya geçmez, sayfayı yenilemez

## NavLink

- Navigasyon için özelleşmiş birkaç özellik içeren bir **Link**
  - activeClassName, activeStyle

# URL'in yapısı



**protocol:** Kaynağa erişmek için kullanılan protokol. (örneğin http, https, ftp vb.).

**hostname:** Kaynağın bulunduğu sunucunun adı. (örneğin www.google.com).

**port:** Kaynağa erişmek için kullanılan port numarası. Bu genellikle belirtilmez ve varsayılan değerler kullanılır (örneğin HTTP için 80, HTTPS için 443).

**path:** Sunucu üzerindeki kaynağın konumu. **(ANA ROUTE)**

**query\_string:** Kaynağa erişirken gönderilen ek bilgiler. Genellikle anahtar-değer çiftlerinden oluşur ve `&` karakteri ile ayrılır (örneğin ?key1=value1&key2=value2).

**fragment\_id:** Kaynağın belirli bir bölümüne doğrudan erişmek için kullanılır. *(bir sayfadaki alt bölümler)*

## useParams hook'u

- URL parametrelerine erişmek için kullanılır.
- Dinamik olarak değişen URL'lerde çok kullanışlıdır.
- Route component'inde path tanımlarken `:` ile tanımlanır. Örn: `:id`
- Bir path'de birden fazla olabilir. Key&value ikilileri olarak bir obje içine aktarılır. useParams hook'u ile erişilebilir. Genelde **destruct** ederek istediğimiz parametre'yi alırız.

```
<Route path="/blog/:slug">
  <BlogPost />
</Route>
```

```
function BlogPost() {
  let { slug } = useParams();
  return <div>Now showing post {slug}</div>;
}
```

# useHistory hook'u

Kullanıcıyı belirli bir eylemi sonrası başka bir sayfaya yönlendirmek için kullanılır.

- **push(path, [state]):** Belirtilen path ve state (isteğe bağlı)'e geçiş yapar.
- **go(n):** Geçmişte n adım ileri veya geri gider. Geri adımlar için negatif değerler kullanılır.
- **goBack():** Geçmiş yığınında bir adım geri gider. go(-1) ile aynıdır.
- **goForward():** Geçmiş yığınında bir adım ileri gider. go(1) ile aynıdır.

```
import { useHistory } from "react-router-dom";

function HomeButton() {
  let history = useHistory();

  function handleClick() {
    history.push("/home");
  }

  return (
    <button type="button" onClick={handleClick}>
      Go home
    </button>
  );
}
```

# Form Elemanları

**Form:** Form elemanlarını grüplamak için kullanılır.

- **onSubmit** özelliği, form gönderildiğinde tetiklenen bir işlevi belirtir. Form verilerini işlemek ve/veya bir sunucuya göndermek için kullanılır.

**Button:** "Type" özelliği, butonun türünü belirtir (örneğin, "submit", "reset" veya "button").

- **onClick** özelliği, butona tıklandığında tetiklenen bir işlevi belirtir.
- Bir form içindeki butonun type özelliği default olarak **submit**'tir.
- Submit butonları tıklandığında form elemanın onSubmit eventini tetikler, **onClick yazmaya gerek yoktur.**

# Form Elemanları

**Textarea:** Kullanıcının metin girişi yapabileceği bir alan oluşturur.

- React'ta, textarea etiketi genellikle bir **name**, **value** ve **onChange** özelliği ile birlikte kullanılır.

**Label:** Bir form elemanı için açıklayıcı metin sağlar.

- "for" özelliği, label'ın hangi form elemanıyla ilişkili olduğunu belirtir. React'da **htmlFor** olarak kullanılır. Form elemanın id'si ile eşleştirilir.
- **htmlFor** yerine **Input**'u **label** tag'lerinin için de yazabiliriz.

# Form Elemanları

**Input:** Kullanıcının veri girişi yapabileceği **tek satırlık** bir alan oluşturur.

- **type** özelliği, giriş alanının türünü belirtir (örneğin, "text", "password", "email" vb.).
- **value** özelliği, giriş alanının mevcut değerini belirtir
- **onChange** özelliği, giriş alanının değeri değiştiğinde tetiklenen bir işlevi belirtir.
- **name** özelliği, alanı tanımlamak için kullanılır. Benzersiz bir isim kullanırız.

# Form ile Çalışmak: Form için state tanımlama

1. Form değerlerini saklamak için obje tipinde bir **state, başlangıç değerleri ile tanımlanır.**

```
const initialValues = {  
  isim: "",  
  soyisim: "",  
  email: "",  
  rol: "",  
};  
  
const [formData, setFormData] = useState(initialValues);
```

# Form Eventleri ile Çalışmak: **onChange**, **onSubmit**

2. Form elemanlarında **onChange** event'i tanımlarız. Value değiştiğinde **state**'i güncelleriz.

```
<div>
  <label className="label" htmlFor="isim">
    Name
  </label>
  <input
    className="form-input"
    onChange={handleOnChange}
    name="isim"
    type="text"
    id="isim"
    value={formData.isim}
  />
</div>
```

```
function handleOnChange(event) {
  const { name, value } = event.target;
  setFormData({ ...formData, [name]: value });
}
```

The diagram shows a code editor interface. On the left, there is an HTML snippet within a `<div>` element. Inside, there is a `<label>` for "Name" and an `<input>` field. The `onChange` attribute of the `<input>` field is highlighted with a yellow box and has an orange arrow pointing to the corresponding `handleOnChange` function definition on the right. The `handleOnChange` function takes an `event` parameter, extracts the `name` and `value` from it, and then updates the `formData` state with the new value for the specified name.

3. Form'a **onSubmit** eventi tanımlarız. State'deki form verilerini kullanarak istenen işlemi gerçekleştiririz.

```
<form onSubmit={submitHandler}>
  ...
</form>
```

```
function submitHandler(event) {
  event.preventDefault();
  axios.post("https://api.wit.com.tr", formData)
    .then(response => {
      console.log(response)
    })
}
```

The diagram shows a code editor interface. On the left, there is an `<form>` element with an `onSubmit` attribute. This attribute is highlighted with a yellow box and has an orange arrow pointing to the `submitHandler` function definition on the right. The `submitHandler` function prevents the default form submission behavior with `event.preventDefault()`. It then uses the `axios` library to make a POST request to the URL `https://api.wit.com.tr` with the `formData` as the payload. The response is logged to the console.

# Controlled Input Kavramı

Bir form element'inin değerini state'teki değere eşitlemek için **value** attribute'una **state'deki değerini** vermeye controlled input denir.

## Neden kullanırız?

- Farklı yerlerden form verisini değiştirmek için (Örn: yazılan değeri büyük harfe çevirebiliriz)
- Form gönderildikten sonra formu başlangıç aşamasına geri getirmek(**reset**) için

```
<div>
  <label className="label" htmlFor="isim">
    Name
  </label>
  <input
    className="form-input"
    onChange={handleOnChange}
    name="isim"
    type="text"
    id="isim"
    value={formData.isim}>
  />
</div>
```

# Advanced Form Elemanları: Select (Dropdown)

Neden tercih edilir?

- Cevabı yazdırma değil seçirmek istiyorsak
- ve olası cevapların sayısı fazlaysa
- ve sayfada çok yer kaplamasını istemiyorsak tercih edilir.
- Örn: Okul, şehir, sokak vb. seçimi

```
<select name="muzik" onChange={handleChange}>
  <option value="rap">Rap</option>
  <option value="rock">Rock</option>
  <option value="metal">Metal</option>
  <option value="pop">Pop</option>
  <option value="arabesk">Arabesk</option>
  <option value="klasik">Klasik</option>
</select>
```

Dikkat:

- **onChange** select'e verilir.
- **Controlled input için:** select value={formData....}

# Advanced Form Elemanları: Radio Button

Neden tercih edilir?

- Cevabı yazdırma değil seçirmek istiyorsak
- ve olası cevapların sayısı fazla değilse tercih edilir.
- Örn: *kayıtlı adreslerden seçim, pizza kenarı seçimi, test cevapları vb.*

Dikkat:

- **İşaretlenmiş mi?** `event.target.checked` ile anlaşılır.
- **Controlled input için:** `input type="radio" checked={...}` (boolean olmalı)
- Tek bir seçenek seçilebilmesi için **name attributeları aynı olmalı**

Uncheck yapılamaz.

```
<p>Favori pet</p>

<label>
  <input
    type="radio"
    onChange={handleChange}
    name="favoriPet"
    value="kedi"
  />
  Kedi
</label>

<label>
  <input
    type="radio"
    onChange={handleChange}
    name="favoriPet"
    value="köpek"
  />
  Köpek
</label>
```

# Advanced Form Elemanları: Checkbox

Neden tercih edilir?

- Cevabı yazdırma değil seçirmek istiyorsak
- ve birden fazla seçim yapılabiliyorsa tercih edilir.
- Örn: *Pizza ekstra malzeme, hobi seçimi vb.*

Dikkat:

- **İşaretlenmiş mi?** `event.target.checked` ile anlaşılır.
  - **Controlled input için:** `input type="checkbox" checked={...}`  
(boolean olmalı)
- Birden fazla seçilebilmesi için **name attributeleri aynı olmalı**

```
<p>Favori meyve</p>

<label className="label-radioCh">
  <input
    type="checkbox"
    onChange={handleChange}
    name="favoriMeyveler"
    value="elma"
  />
  Elma
</label>

<label className="label-radioCh">
  <input
    type="checkbox"
    onChange={handleChange}
    name="favoriMeyveler"
    value="armut"
  />
  Armut
</label>
```

# Checkbox'la Çalışmak: State İşlemleri

State'de Array olarak saklanır.(Ekleme ve çıkarma işlemleri kolay)

## 1. State'i nasıl güncelleriz?

- Eleman değiştiğinde ([onChange](#))
- değişen eleman checkbox ise:  **event.target.type === "checkbox"**
- State içinde var mı?
  - Var. Demek ki çıkarılmalı: 
  - Yok. Demek ki eklenmeli: 

```
let newValue;

// tıklanan eleman checkbox mı?
if (event.target.type === 'checkbox') {
  // state içinden ilgili bölümü al.
  const oldValues = formData[event.target.name];

  // tıklanan checkbox değeri state içinde var mı?
  if (oldValues.includes(event.target.value)) {
    // var, çıkar
    newValue = oldValues.filter((v) => v !== event.target.value);
  } else {
    // yok, ekle
    newValue = [...oldValues, event.target.value];
  }
} else {
  // tıklanan eleman checkbox değil, normal işlem
  newValue = event.target.value;
}
```

# Ekstra: Çok Sayıda Checkbox'la Çalışmak

1. Checkbox bilgilerini objelerden oluşan bir array'de sakla.

```
const meyveler = [  
  {  
    label: 'Muz',  
    value: 'muz',  
  },  
  {  
    label: 'Çilek',  
    value: 'çilek',  
  },  
];
```

## Ekstra: Çok Sayıda Checkbox'la Çalışmak

2. Checkbox için component oluştur. Prop olarak

- isChecked
- name
- info: label, value
- onChangeFn

```
export default function CheckBox({ onChangeFn,  
isChecked, info, name }) {  
  return (  
    <label className="label-radioCh">  
      <input  
        type="checkbox"  
        onChange={onChangeFn}  
        checked={isChecked}  
        name={name}  
        value={info.value}>  
      />  
      {info.label}  
    </label>  
  );  
}
```

## Ekstra: Çok Sayıda Checkbox'la Çalışmak

3. .map ile tüm checkbox'ları listele.

```
{meyveler.map((meyve) => (
  <CheckBox
    isChecked={formData.favoriMeyveler.includes(meyve.label)}
    onChangeFn={handleChange}
    info={meyve}
  />
))}
```

# Validation Nedir?

Kullanıcıların girmiş olduğu verilerin, veritabanına kaydedilmeden önce istenen şartları sağlayıp sağlamadığını kontrol etmeye denir. Örn:

- Kullanıcı, doğru formatta *email adresi* girmiş mi? *egitim@gmaill.com* yerine *egitim@gmail.com*
- Tarihi, beklenen formatta yazmış mı? *22/02/1978* yerine *22.02.1978*

2 Farklı yerde validation yapabiliriz: Client side validation, server side validation

- Client'ta yaparsak server yükü azalır
- Kullanıcıyı bekletmemiş oluruz
- Formu tekrar doldurması gerekmez

# Validation Aşamaları

1. Validation için gerekli **2 state'i ve hata mesajları objesini oluştur**: **errors**(object), **isValid**(boolean), **errorMessages**(object)

```
const [errors, setErrors] = useState({  
  name: false,  
  email: false,  
  terms: false,  
});  
  
const [isValid, setIsValid] = useState(false);
```

```
const errorMessages = {  
  name: "İsim alanı en az 4 karakter olmalıdır.",  
  email: "Geçerli bir email adresi yazınız.",  
  terms: "Kayıt olmak için anlaşma şartlarını kabul etmelisiniz."  
}
```

# Validation Aşamaları

- handleChange fonksiyonu içinde **validasyonları yap** ve sonucuna göre **errors state'ini güncelle**.

```
const handleChange = (event) => {
  let { type, name, checked, value } = event.target;
  value = type == 'checkbox' ? checked : value;
  setForm({ ...form, [name]: value });

  if (name == 'email') {
    if (validateEmail(value)) {
      setErrors({ ...errors, [name]: false });
    } else {
      setErrors({ ...errors, [name]: true });
    }
  }

  if (name == 'name') {
    if (value.replaceAll(' ', '').length >= 4) {
      setErrors({ ...errors, [name]: false });
    } else {
      setErrors({ ...errors, [name]: true });
    }
  }
};
```

# Validation Aşamaları

- useEffect'de form valid kontrolü yap.

```
useEffect(() => {
  if (
    validateEmail(form.email) &&
    form.name.replaceAll(' ', '').length >= 4 &&
    form.terms
  ) {
    setIsValid(true);
  } else {
    setIsValid(false);
  }
}, [form]);
```

# Validation Aşamaları

4. Hata mesajlarını göster.

```
<p className="error">{errors.name && errorMessages.name}</p>
```

5. Buton'u form valid değil iken **disabled** yap.

```
<button disabled={!isValid} type="submit">  
| Kaydet  
</button>
```

# Verileri Göndermek

Eğer **form valid ise** verileri;

- handleSubmit fonksiyonu içinde
- **axios.post** ile gönderebiliriz.
- unutma!
  - o event.preventDefault()
  - o isValid kontrolü

- Deneme yapmak için: <https://regres.in/>

```
const handleSubmit = (event) => {
  event.preventDefault();
  if(!isValid) return;

  axios.post(url, formData)
    .then(response=>{
      history.push("/")
    })
    .catch(error=>{
      console.warn(error);
      history.push("/errorPage")
    })
}
```

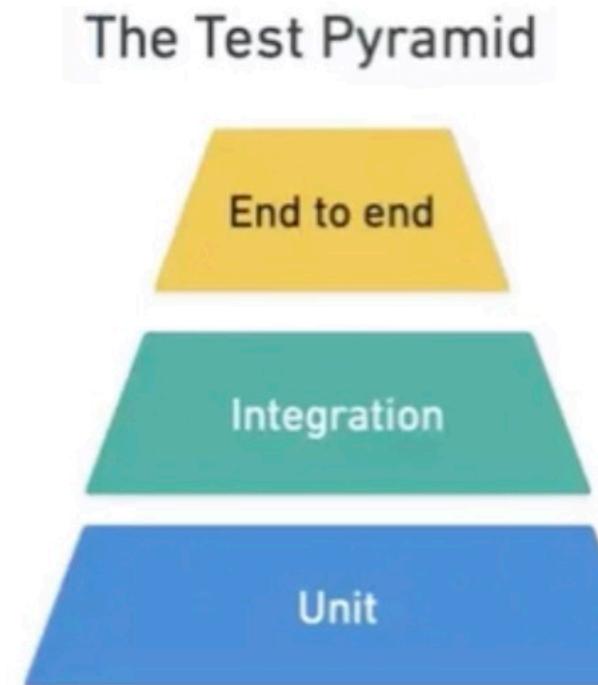
# Kod Testleri

Kod testleri, yazdığımız kodun doğru çalışıp çalışmadığını kontrol etmek için kullanılır.

- Çok yazılımcının birlikte çalıştığı projelerde hayatidır.
- Herkes yazdığı kodu test etse bile, diğerlerinin yazdığı kodla olan uyumunu test edemez, çok uzun sürer.
- Manuel testler; zor, maliyetli, insan faktörü hatalara müsait
- Bu yüzden test kütüphaneleri kullanırız

4 farklı test vardır:

1. Statik testler
2. Unit testler
3. Entegrasyon testleri
4. E2E testleri



# 1. Static Test

Kodun çalıştırılmadan kontrol edildiği bir test türüdür.

- Kodun okunabilirliği, düzgün yapılandırılması ve belirli standartlara uygunluğu bu test ile kontrol edilir.
- Hataların erken aşamada tespit edilmesi ve düzeltilmesi amaçlanır.
- Kod yazımı ve revizyon aşamalarında kullanılır.
- **Amaç, kodun kalitesini artırmak ve gelecekteki hataları önlemektir.**
- ESLint

## 2. Unit Test

Yazılımın en küçük parçası olan **birimlerin** (fonksiyonlar, prosedürler, sınıflar) ayrı ayrı test edildiği bir test türüdür.

- Her birim, belirlenen girdilere karşı beklenen çıktıları verip vermediği kontrol edilir.
- Kodun herhangi bir yerinde değişiklik yapıldığında kullanılır.
- **Amaç, her birimin doğru çalıştığını doğrulamak** ve hataları erken aşamada tespit etmektir.

### 3. Integration Test

Birim testlerden geçmiş kod parçalarının bir araya getirilerek bir **bütün olarak çalışıp çalışmadığının** kontrol edildiği bir test türüdür.

- Farklı birimlerin birlikte çalışırken doğru veri alışverişi yapması ve beklenen işlevleri yerine getirmesi kontrol edilir.
- Birimlerin birleştirilmesi aşamasında kullanılır.
- **Amaç, birimlerin birlikte doğru çalıştığını doğrulamak** ve birleştirme hatalarını tespit etmektir.

## 4. End to End (E2E) Test

Kullanıcının perspektifinden yazılımın tüm **bileşenlerinin birlikte çalışıp çalışmadığının** kontrol edildiği bir test türüdür.

- Gerçek kullanıcı senaryoları kullanılarak **yazılımın tüm katmanları** test edilir.
- Yazılımın kullanıma sunulmasından önce kullanılır.
- **Amaç, yazılımın gerçek dünya koşullarında doğru çalıştığını doğrulamak** ve kullanıcı deneyimini iyileştirmektir.

# Test Mühendisi (QA) gibi düşünmek: Kod testi mantığı kurmak

- Kod testi mantığı, yazılımın belirli bir işlevi yerine getirip getiremediğini kontrol etmeye dayanır.
- Test mühendisi gibi düşünmek, bir nevi **dedektiflik yapmak gibidir**. Yazılımın hangi durumlarda hata verebileceğini düşünüp, bu hataları bulmaya çalışırız.



Bill Sempf  
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

# Testin 3 Aşaması: **Arrange-Act-Assert**

Arrange-Act-Assert, bir kod testini düzenlemek ve yazmak için kullanılan bir yöntemdir.

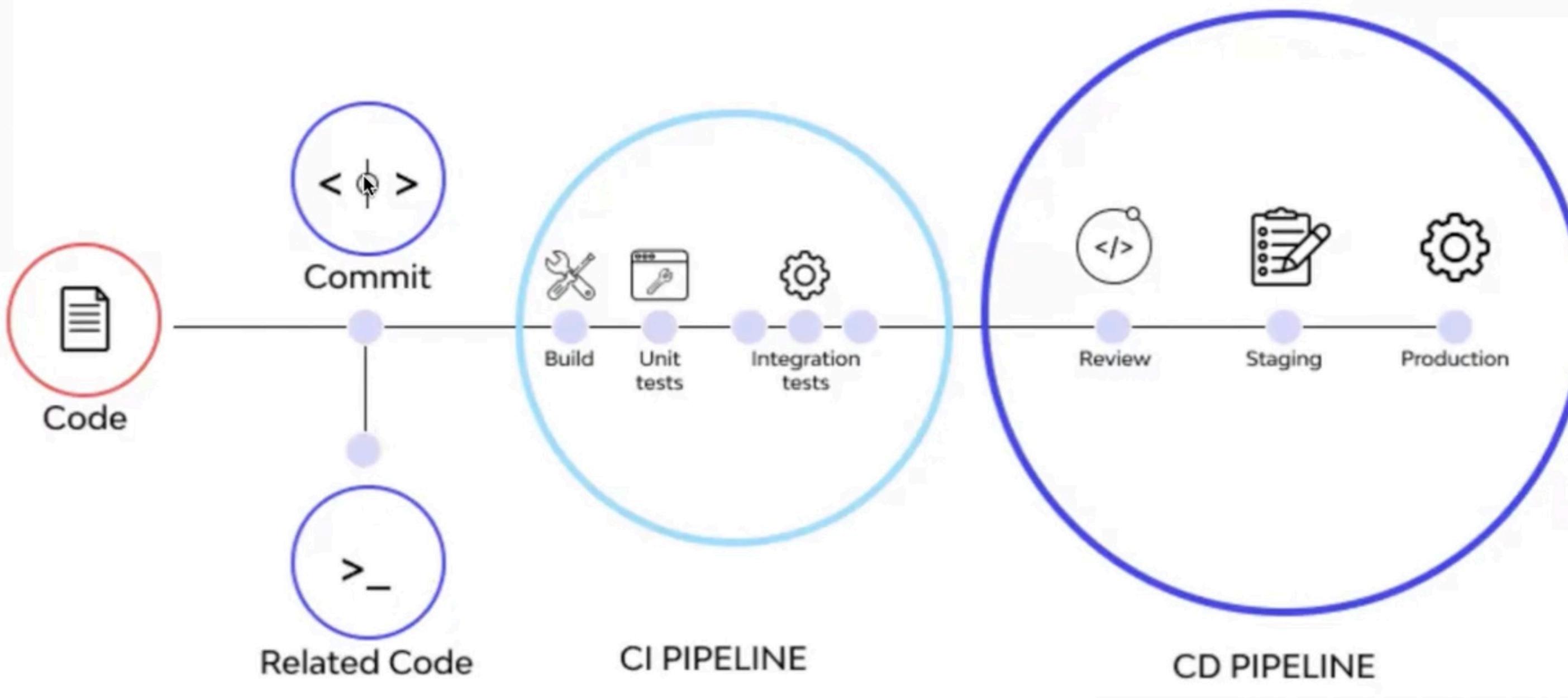
- **Arrange** aşamasında, testin çalışması için gerekli tüm ayarları yapıyoruz.
  - Örn: *uygulamanın veritabanını sıfırlayıp*
  - *uygulama linkini açıp*
  - *sign up butonuna basmak*
- **Act** aşamasında, test edilecek işlemi gerçekleştiriyoruz.
  - Örn: *formdaki ad input'unu yakalamak*
  - *içine isim yazmak*
  - *formu doldurmak ve sign up butonuna basmak*
- **Assert** aşamasında ise, işlemin sonucunun beklediğimiz gibi olup olmadığını kontrol ediyoruz.
  - Örn: *kullanıcı kayıt oldu mu*
  - *ana sayfaya yönlendi mi*
  - *kullanıcı ismiyle hoş geldin mesajı çıktı mı?*

## **Ekstra: CI/CD (Continuous Integration/Continuous Deployment) pipeline**

Yazılım geliştirme sürecini otomatikleştiren bir yaklaşımdır.

- **CI**, geliştiricilerin kodlarını paylaştığı merkezi bir depoya sürekli olarak birleştirmelerini ifade eder. Bu, kod tabanının sürekli olarak güncel ve tutarlı kalmasını sağlar.
- **CD**, kodun sürekli olarak test edilmesini ve üretime dağıtılmasını ifade eder. Bu, yazılımın sürekli olarak güncel ve kullanılabilir kalmasını sağlar.
- **Kodlar testten geçmezse son kullanıcılar hatalı kodu görmez.**

# **Ekstra: CI/CD (Continuous Integration/Continuous Deployment) pipeline**





## Cypress.io <https://www.cypress.io/>

Tüm test türlerini yazabileceğimiz görsel bir test kütüphanesidir.

- Testler **istediğimiz browserda** açılır ve tüm test akışını takip edebiliriz
- Asenkron işlemlere uygundur
- Elemanları seçmek - bazı eventleri tetiklemek - sonuçları kontrol etmek için kullanabiliriz.
  - Örn: *Giriş formunda inputları seçmek*
  - *içine kullanıcı bilgisi yazmak*
  - *giriş yapıyor mu kontrol etmek*

# Cypress.io Kullanımı

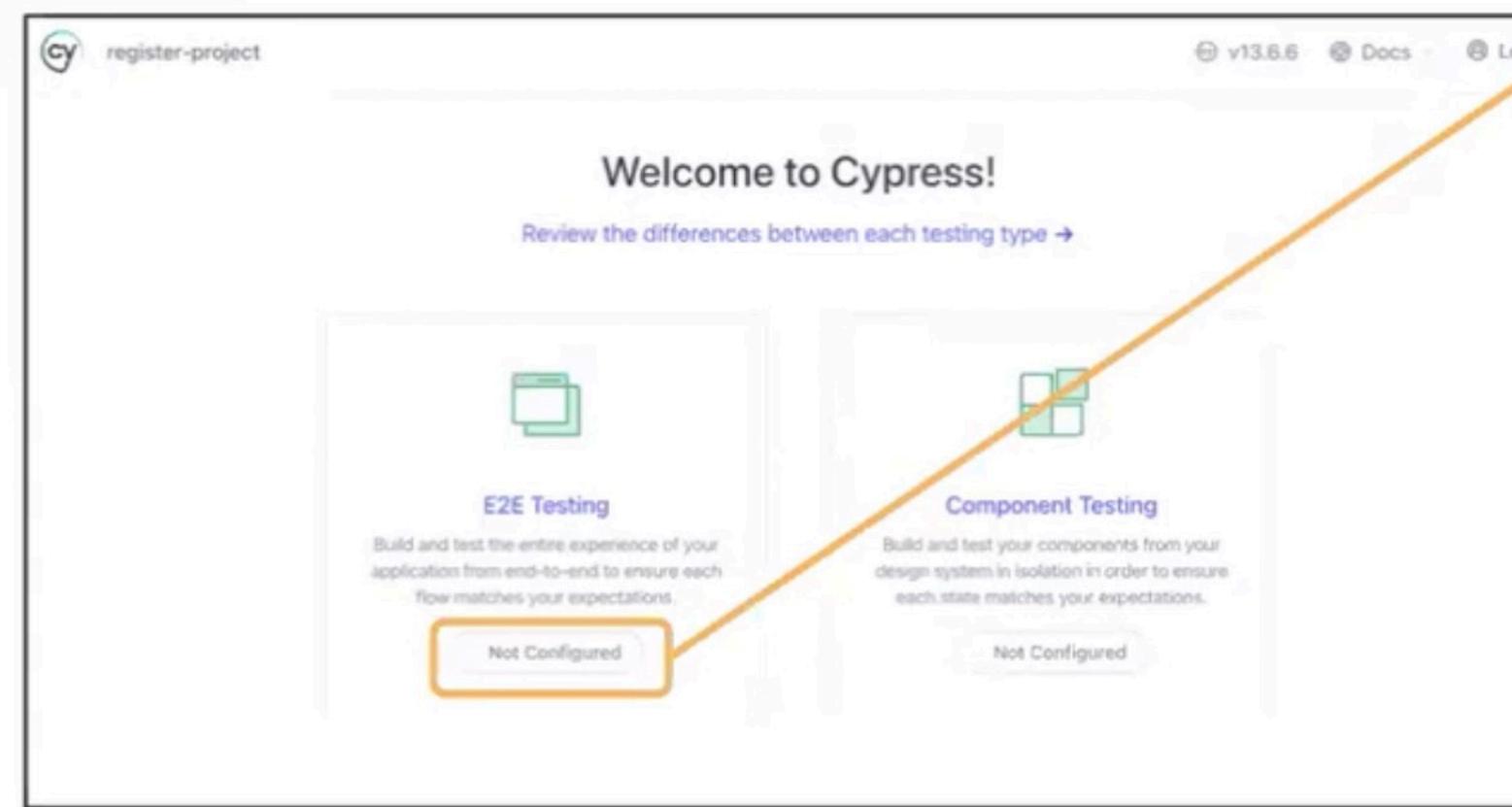
1. projeye cypress kütüphanesi yüklenir.

```
› npm i -D cypress
```

2. cypress arayüzü açılır.

```
› npx cypress open
```

3. kurulum adımları yapılır.

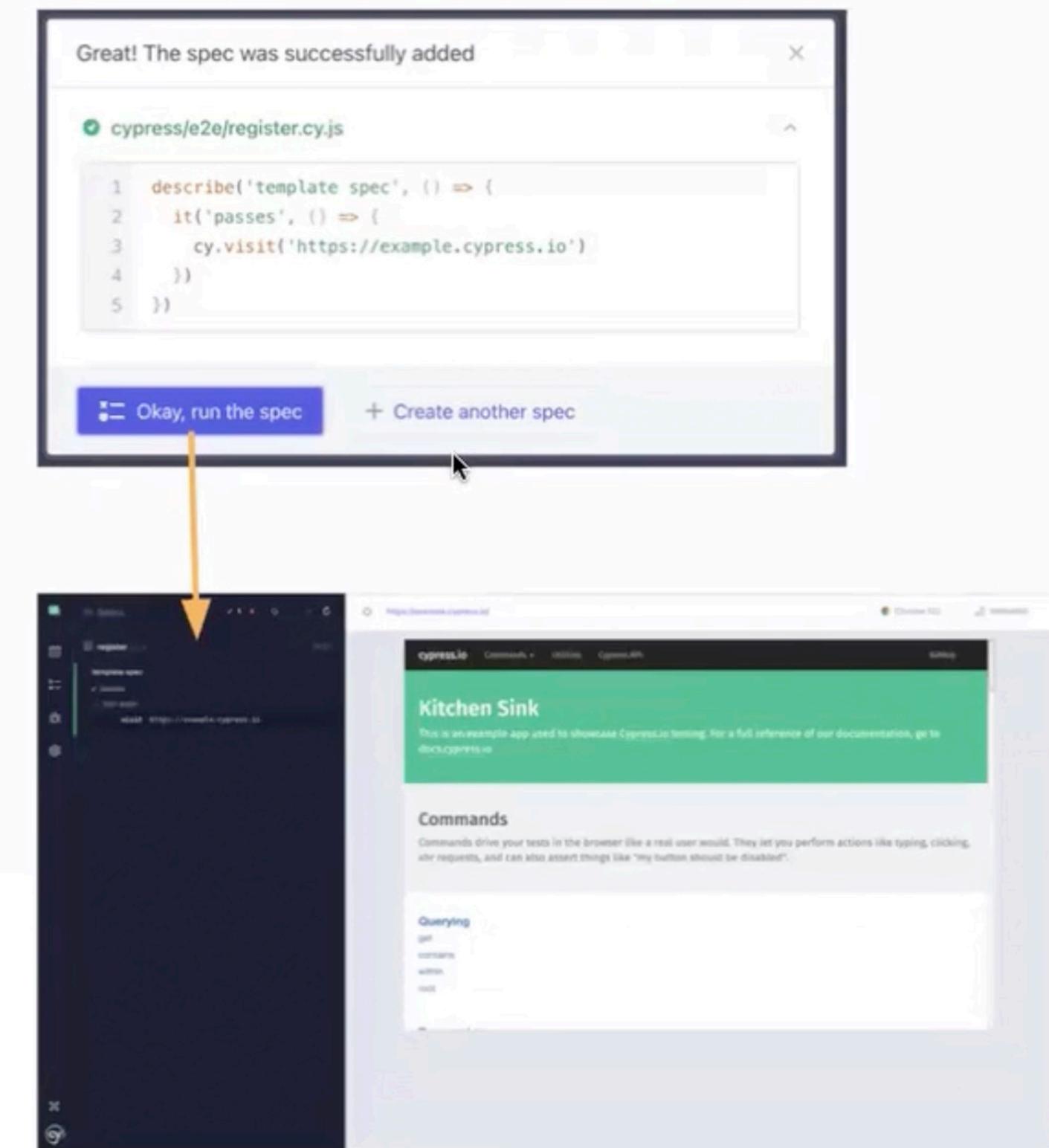
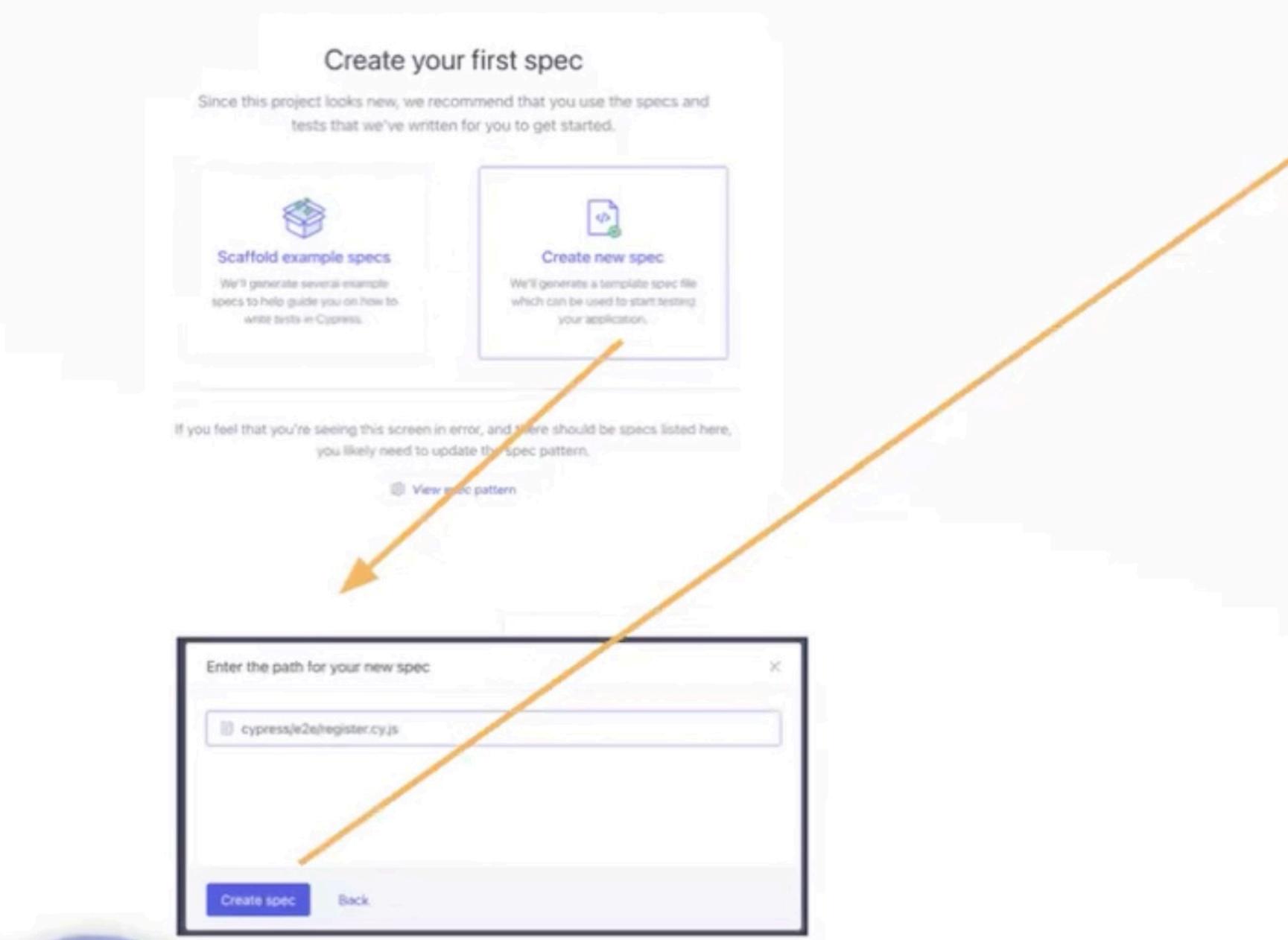


This screenshot shows the 'Configuration files' setup screen. It lists four files added to the project: 'cypress.config.js', 'cypress/support/e2e.js', 'cypress/support/commands.js', and 'cypress/fixtures/example.json'. Each file has a brief description. A blue 'Continue' button is at the bottom. An orange arrow points from the 'Continue' button towards the 'Choose a browser' screen.

This screenshot shows the 'Choose a browser' screen. It allows selecting a preferred browser for E2E testing, with 'Chrome v92' selected. Other options include 'Electron v116'. A green 'Start E2E Testing in Chrome' button is at the bottom. An orange arrow points from the 'Start E2E Testing in Chrome' button back towards the E2E Testing section of the previous screen.

# Cypress.io Kullanımı

4. yeni bir test dosyası oluşturuldu ve testler yazılır.



# Cypress.io'da Test Yazma

```
import React, { useState } from "react";

function App() {
  const [message, setMessage] = useState("");

  const handleClick = () => {
    setMessage("Merhaba, Cypress!");
  };

  return (
    <div>
      <button onClick={handleClick}>Tıkla</button>
      <p>{message}</p>
    </div>
  );
}

export default App;
```

```
// test tanımı
describe('My First Test', () => {
  // test bölümünün adı
  it('finds the content "type"', () => {
    // Uygulamanın çalıştığı adresi ziyaret et
    cy.visit("http://localhost:3000");
    // 'Tıkla' yazısını içeren bir elementi bul ve tıkla
    cy.contains("Tıkla").click();
    // 'Merhaba, Cypress!' yazısını içeren bir elementi bul
    cy.contains("Merhaba, Cypress!");
  });
})
```

Arrange

Act

Assert

# Cypress.io'da Eleman Seçme

Selector	Recommended	Notes
<code>cy.get('button').click()</code>	⚠ Never	Worst - too generic, no context.
<code>cy.get('.btn.btn-large').click()</code>	⚠ Never	Bad. Coupled to styling. Highly subject to change.
<code>cy.get('#main').click()</code>	⚠ Sparingly	Better. But still coupled to styling or JS event listeners.
<code>cy.get(' [name="submission"] ').click()</code>	⚠ Sparingly	Coupled to the <code>name</code> attribute which has HTML semantics.
<code>cy.contains('Submit').click()</code>	✓ Depends	Much better. But still coupled to text content that may change.
<code>cy.get(' [data-cy="submit"] ').click()</code>	✓ Always	Best. Isolated from all changes.

# Cypress.io'da Event Tetikleme [Cypress Actionability Guide](#)

## Actionability

Some commands in Cypress are for interacting with the DOM such as:

- `.click()`
- `.dblclick()`
- `.rightclick()`
- `.type()`
- `.clear()`
- `.check()`
- `.uncheck()`
- `.select()`
- `.trigger()`
- `.selectFile()`

```
cy.get('.btn').click()
```

# Cypress.io'da Assertion [Cypress Assertion Reference](#)

## Common Assertions

Here is a list of common element assertions. Notice how we use these assertions (listed above) with `.should()`. You may also want to read about how Cypress [retries](#) assertions.

### Length

```
// retry until we find 3 matching <li.selected>
cy.get('li.selected').should('have.length', 3)
```

### Class

```
// retry until this input does not have class disabled
cy.get('form').find('input').should('not.have.class', 'disabled')
```

### Value

```
// retry until this textarea has the correct value
cy.get('textarea').should('have.value', 'foo bar baz')
```