JavaScript POST-metod

19 OKTOBER 2024

JavaScript – utan tjafs
Författare: Acke Strömberg

För en kanin

POST-metod

Introduktion

När man behöver hämta data till sin webbapplikation genom ett REST API så gör man det med en asynkron funktion som heter fetch. Det är enklast att visa genom ett exempel. Vi skissar upp en enkel webbapplikation som visar personer med namn och lite annan information. Se bild 1 för HTML-kod, bild 2 för CSS-kod och bild 3 för den JavaScript-kod som behövs för den här applikationen.

```
<!DOCTYPE html>
<html lang="sv">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="style.css" />
    <title>JavaScript - Fetch-anrop</title>
  </head>
  <body>
    <h1>JavaScript - Fetch-anrop</h1>

    <button>Hämta</button>

    <h2>Alla i gruppen</h2>

    <ul class="cards">
      <li class="card">
        <h3 class="card-namn">Kalle Kanin</h3>
        <div class="card-beskrivning">
          <p>Kalle är en pigg och glad kanin</p>
        </div>
      </li>

      <li class="card">
        <h3 class="card-namn">Nisse Snubbelfot</h3>
        <div class="card-beskrivning">
          <p>Nisse är glad men något klumpig.</p>
        </div>
      </li>
    </ul>

    <script src="main.js" defer></script>
  </body>
</html>
```

Bild 1 HTML-filens utseende i början.

```
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  max-width: 960px;
  margin: 2em auto 0;
  font-family: Arial;
  text-align: center;
}

h1,
h2 {
  margin-bottom: 1em;
}

button {
  background-color: #04aa6d;
  border: none;
  color: white;
  padding: 15px 32px;
  margin-bottom: 1em;
  font-size: 1em;
}

.cards {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-gap: 20px;
  max-width: 960px;
  margin: 0 auto;
}

.card {
  border: 1px solid black;
  min-width: 200px;
  display: flex;
  flex-direction: column;
  padding: 1em 2em;
}

.card-namn {
  margin-bottom: 1em;
}
```

Bild 2 CSS-filens utseende i början.

```
"use strict"

const knapp = document.querySelector("button")
const kaninRubrik = document.querySelector("h2")
const kaninLista = document.querySelector("#kanin-lista")

function main() {
  kaninRubrik.style.display = "none"
  kaninLista.style.display = "none"

  console.log("allting funkar")
}

main()
```

Bild 3 JavaScript-filens utseende i början.

Om man drar igång HTML-filen med hjälp av Live Server-tillägget i Visual Studio, så ser man en lite fjuttig webbsida med endast en rubrik och en knapp där det står Kaniner. Tanken är nu att vi ska hämta namn på kaniner från ett REST API och fylla lite kort som visar dessa på hemsidan. I filen main.js finns det en funktion som heter main(). Kommentera bort raderna med display = 'none' så kan ni se hur sidan kommer att se ut när vi har hämtat informationen. Nu låter vi dessa två rader vara med i programmet igen och då bör vi bara se rubriken och knappen igen.

Det vi vill ska hända nu är att när man trycker på knappen så gör applikationen ett HTTP-anrop till det REST API som används här. Det är https://jsonplaceholder.typicode.com/users , och nu får vi alltså låtsas att det är listor av kaniner som vi hämtar, inte användare.

Händelselyssnare

Vi behöver koppla en event listener till knappen så att vi vet när vi ska hämta data. Det gör vi genom att skriva kommandot enligt bild 4.

```
knapp.addEventListener("click", hamtaKaniner)

function hamtaKaniner() { }
```

Bild 4 Lägger till en eventListener till knappens click event.

När man nu klickar på knappen så kommer eventListener se till att funktionen hamtaKaniner() anropas. Vi testar att lägga in console.log("hej") i den funktionen.

Om vi öppnar developer tools i webbläsaren med F12 så bör vi se hej i konsolen när vi klickar på knappen.

Anonyma funktioner

Det är mycket vanligt att jobba med något som kallas anonymous functions (anonyma funktioner) när man använder metoden fetch() i JavaScript. Det är alltså frågan om funktioner som inte har ett namn och därför inte kan anropas från andra platser i applikationen förutom där programmet just nu befinner sig. På bild 5 och 6 så visas exakt samma hämtning av kaniner, men på bild 5 används traditionella anonymous functions och på bild 6 något som kallas för fat arrow functions (fetpilsfunktioner) eller bara arrow functions. Det ser ut som en fet pil, men är ett modernare skrivsätt för att visa att det är en funktion.

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function(response) {
    return response.json()
  })
  .then(function(data) {
    console.log(data)
  })
```

Bild 5 Användandet av anonymous functions.

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then((response) => response.json())
  .then((data) => console.log(data))
```

Bild 6 Användandet av fat arrow functions.

Som man kan se blir det betydligt lättare att läsa den koden enligt bild 6. Eftersom sättet att skriva enligt bild 6 också är det vanligaste förekommande så är det också vad som kommer att visas här.

Själva hämtet…

Vi kommer att göra ett GET-anrop till REST API:et och skrivsättet för det är enligt bild 7.

```
      function hamtaKaniner() {
1 =>    fetch("https://jsonplaceholder.typicode.com/users")
2 =>      .then(function(response) => response.json())
3 =>      .then((data) => console.log(data))
      }
```

Bild 7 Den enklaste typen av fetch-anrop.

Det är nu det blir lite speciellt med den asynkrona koden. Man behöver dock inte förstå allt i detalj, men däremot känna till vad metoden fetch() skickar tillbaka som svar när man använder den. Den skickar tillbaka något som kallas Promise. Det är ett begrepp som förekommer i JavaScript när man använder asynkrona funktioner bland annat. Ett så kallat Promise måste man ta hand om och göra någonting med annars blir JavaScript-körningen konfunderad och börjar krångla.

Om man tittar på andra raden i funktionen hamtaKaniner() så börjar den raden med en punkt och sen kommer en harang. Den där punkten visar egentligen bara att den hör ihop med uttrycket på raden ovanför, den där fetch-metoden som alltså skickar tillbaka ett Promise. Det skulle gå lika bra att skriva rad 1, 2 och 3 i funktionen hamtaKaniner() på en enda lång rad. Problemet är bara att det är lite opraktiskt och därför delar man nästan alltid upp sådana långa rader med en punkt som börjar på raden nedanför.

På rad 2 är det .then() som tar hand om det första Promise som fetch() skickar tillbaka. Det kallas här för response och är alltså det svar man får från fetch(). Man får kalla det precis vad man själv vill, det är dock vanligt att skriva res, resp eller response. Var gärna så tydlig som möjligt för er egen skull. Vill ni kalla det för svar så går det alldeles utmärkt. Tänk då bara på att det sista uttrycket på rad 2 då blir svar.json(). Det som händer i slutet på rad 2 är alltså att man tar svaret som kommer tillbaka från fetch-metoden och anropar metoden json() på den. Det är för att kunna göra om det svaret till någonting som är läsligt. Metoden json() returnerar också ett Promise som vi tar hand om på rad 3. Där har vi kallat det vi får tillbaka från response.json() för data. Men man får kalla det för precis vad man vill också. Vill man kalla det för kaninInformation så går det också bra. Sista uttrycket på rad 3 blir då console.log(kaninInformation).

Vad är det vi har hämtat?

Allt är bra och vi är lyckliga, men vi har inte en aning om vad det är vi har gjort. Vi behöver titta närmare på vad vi egentligen har ställt till med. Om allt är rätt och vi har lyckats göra ett fetch-anrop så bör vi kunna se ett resultat i webbläsarens developer tools. Öppna fliken Console eller Konsol så bör man se en utskrift som ser ut ungefär som på bild 8.

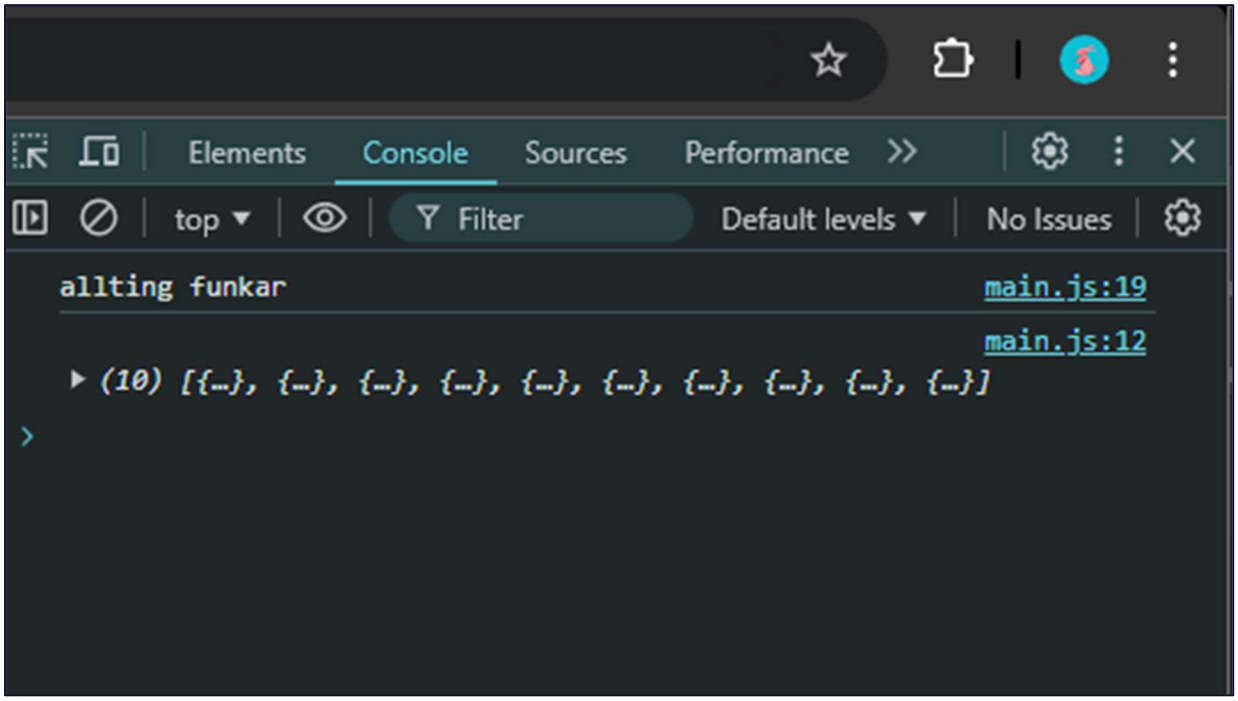

Bild 8 Utskrift av listan med objekt som vi har hämtat.

Det är den under utskriften som är den intressanta. Först en högerpil, sen en parentes med talet 10 och sen en massa måsvingar och hakparenteser. Vad är nu det här? För att begripa det här behöver man klicka på högerpilen och vips får man 10 rader som är numrerade från 0 till 9. Den översta raden sa alltså att vi hade en lista eller array med 10 objekt i sig. Se närmare på bild 9.

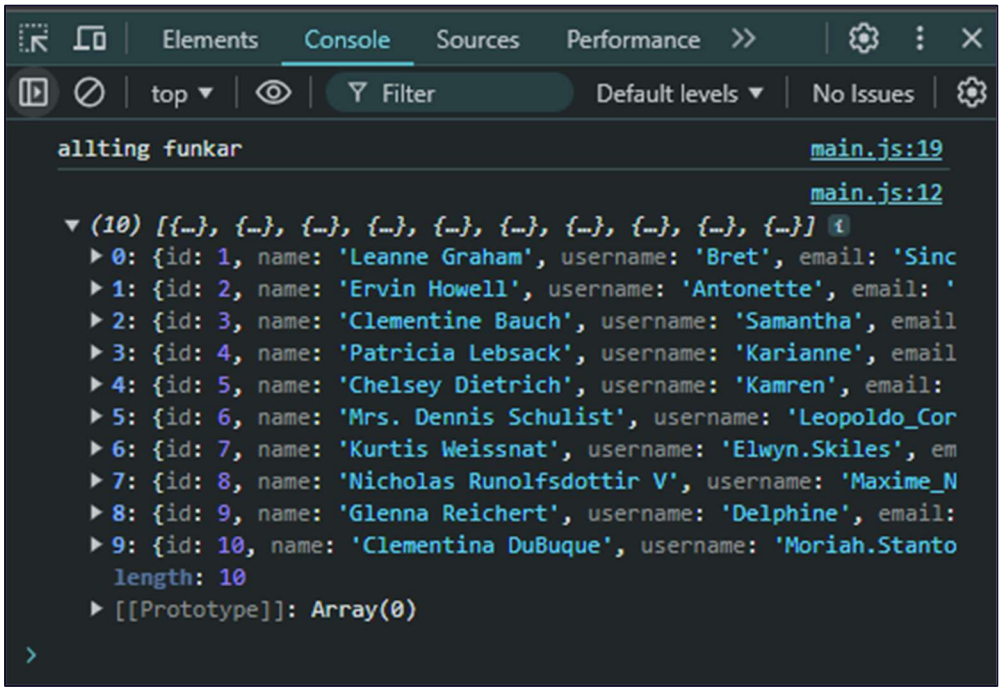

Bild 9 Varje objekt i listan, rad för rad.

Det finns ett par saker att notera här. Det är första är talet 10 i parentes på översta raden. Det säger att det finns så många objekt i listan. Att det är en lista ser vi dels genom att det står Array under alla objekt, men framför allt på den översta raden. Efter talet 10 i parentes kommer en hakparentes och flera måsvingar med tre punkter mellan sig.

Om vi går vidare och klickar på den högerpil som är framför 0 i det första objektet, så får vi ytterligare lite mer information. Bild 10 är det första objektet i detalj med flera olika attribut och underattribut.

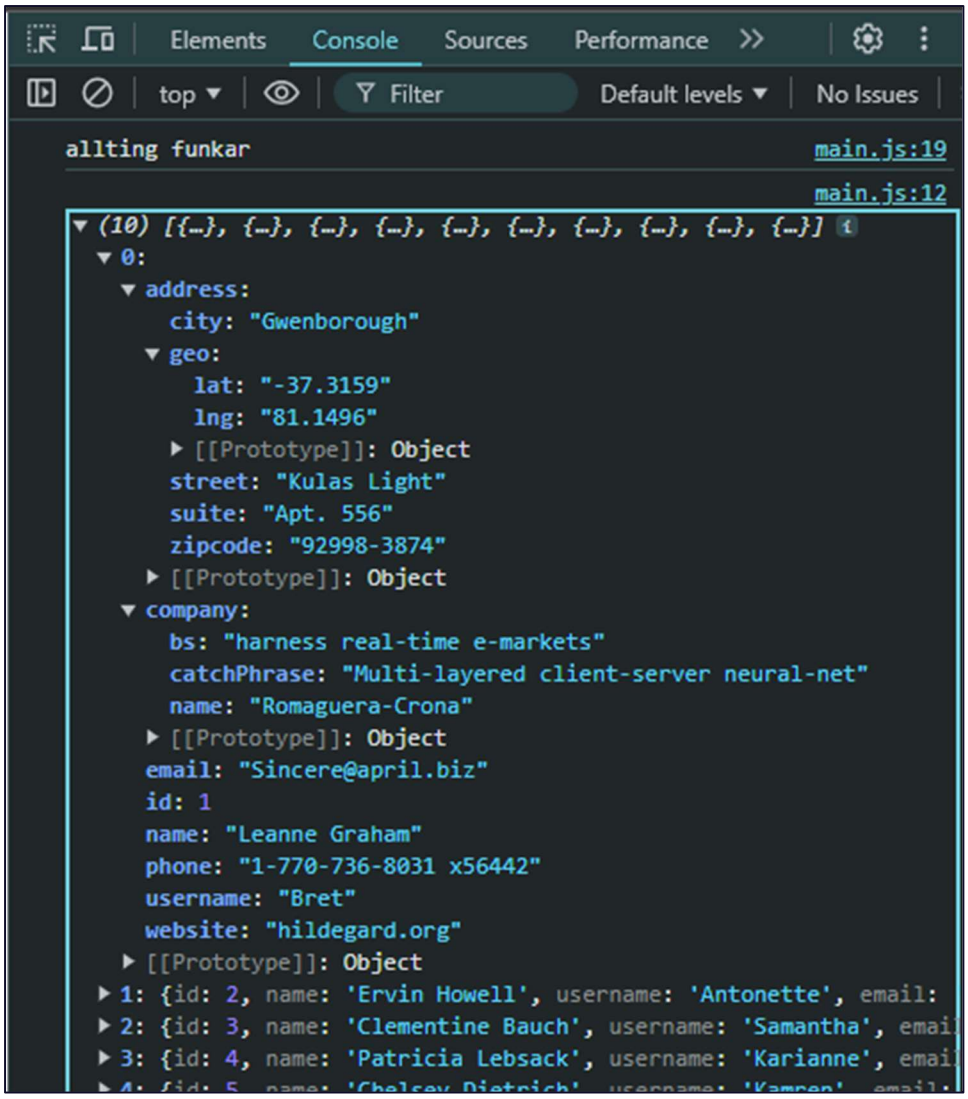

Bild 10 Första objektet i detalj, med alla attribut.

Vi har alltså fått tillbaka en lista med 10 objekt, där varje objekt har åtta huvudattribut. Det är address, company, email, id, name, phone, username och website. Sen har address och company i sin tur underattribut. Lite konstigt med kaniner som har ett telefonnummer och egen website, men det är verkligen speciella kaniner det här.

Skapa kort med information

Nu är det dags gå vidare och göra något konstruktivt med all vår data från den här listan som vi har hämtat. Vi tar den data vi har hämtat och skickar den till en ny funktion - skapaKaniner(). Det första vi gör är att testa med en utskrift för att se att vi verkligen kommer till den funktionen, enligt bild 11.

```
function skapaKaniner(kaninData) {
  console.log("--> kaninData", kaninData)
}

function hamtaKaniner() {
  fetch("https://jsonplaceholder.typicode.com/users")
    .then((response) => response.json())
    .then((data) => {
      skapaKaniner(data)
    })
}
```

Bild 11 En console.log i funktionen skapaKaniner

I funktionen hamtaKaniner() ser man anropet till skapaKaniner() där vi skickar med datat som vi har fått från REST API:et. Vi kallar den parametern för kaninData och gör en utskrift av den för att kontrollera att det verkligen fungerar. Nu behöver vi i funktionen först visa rubriken och själva listan. Det kan vi göra enligt bild 12.

```
function skapaKaniner(kaninData) {
  kaninRubrik.style.display = "block"
  kaninLista.style.display = "block"
}
```

Bild 12 En console.log i funktionen skapaKaniner

Nu borde listan med Kalle Kanin och Nisse Snubbelfot synas igen. Det andra vi vill göra är att stega igenom den lista vi har fått med oss, post för post och skapa ett nytt listelement för varje post. Där vill vi visa namnet på kaninen och någon fras från det objektet. Vi bestämmer vad när vi kommer dit. För att gå igenom en lista i JavaScript så är det bra att använda någon av de redan färdigskrivna funktionerna. Här är forEach() ett utmärkt val. Då skriver man enligt bild 13.

```
kaninData.forEach((kanin) => {
  console.log(kanin)
})
```

Bild 13 En console.log i funktionen skapaKaniner

Det här är precis som en vanlig for-loop, men det är ett skrivsätt som vi föredrar. Inne i själva funktionen kommer varje listelement att vara tillgängligt för oss. Vi kan kalla det vad vi vill, men här kallar vi varje post för kanin. Efter fetpilen kommer måsvingar och innanför dessa kan vi utföra instruktioner som vi vi göra med varje kanin. Vi vill hämta kaninens namn och sen vill vi också hämta ett attribut som ligger under company som heter catchPhrase, se bild 10. Det kan man hämta genom att skriva kanin.name och kanin.company.catchPhrase. I for-loopen måste vi skapa ett li-element, ett h3-element, ett div-element och ett p-element, och dessutom förse dessa med lämpliga attribut. Det går att göra på två olika sätt. Vi går igenom båda dessa skrivsätt, därefter får du själv välja vilket som passar dig. Skrivsätt 1 är enligt bild 14 och skrivsätt 2 enligt bild 15.

```
kaninData.forEach((kanin) => {
  const li = document.createElement("li")
  li.setAttribute("class", "card")

  const h3 = document.createElement("h3")
  h3.setAttribute("class", "card-namn")
  h3.innerText = kanin.name

  const div = document.createElement("div")
  div.setAttribute("class", "card-beskrivning")

  const p = document.createElement("p")
  p.innerText = kanin.company.catchPhrase

  div.appendChild(p)
  li.appendChild(h3)
  li.appendChild(div)
  kaninLista.appendChild(li)
})
```

Bild 14 Lägga till listelement enligt skrivsätt 1.

Enligt bild 14 måste man först skapa ett li-element och lägga till ett class-attribut, därefter skapa ett h3-element, lägga till ett class-attribut och innerText. Sedan går man vidare med ett div- och p-element och lägger till det som behövs. Det sista steget är att man får gå inifrån och ut genom att först appendChild av p- till div-elementet, h3- och div-elementet till li-elementet och slutligen lägga till li-elementet till kaninLista. Det var lite jobb men det går kanske att hänga med på arbetsgången?

1. Skapa respektive element
2. Lägg till attribut och text
3. Koppla varje element till sin förälder

```
kaninData.forEach((kanin) => {
  kaninLista.innerHTML += `<li class="card">
     <h3 class="card-namn">${kanin.name}</h3>
     <div class="card-beskrivning">
       <p>${kanin.company.catchPhrase}</p>
     </div>
    </li>`
});
```

Bild 15 Lägga till listelement enligt skrivsätt 2.

Skrivsätt 2 är beskrivet i bild 15, ovan och är betydligt smidigare. För varje post i listan så lägger vi till nytt innehåll till html-elementet kaninLista, genom frasen kaninLista.innerHTML. Vi vill behålla de list-element vi redan har lagt till och måste därför skriva += . Bild 16 förklarar skrivsättet.

```
// Skrivsättet
kaninLista.innerHTML += ’<li class="card">’

// Motsvarar
kaninLista.innerHTML = kaninLista.innerHTML + ’<li class="card">’
```

Bild 16 Förklaring av skrivsättet +=.

Efter += kommer något som kallas för backticks, alltså en bakåt apostrof. Det tecknet får man om man håller ner shift och knappen till vänster om Backspace. Det är också två knappar till höger om siffran 0 på tangentbordet. Lite knepigt, men du ska alltså hålla ner shift och apostrof-knappen, då händer det ingenting. Men om du nu släpper dessa knappar och trycker på mellanslag så kommer det en backtick. Shift + apostrof => släpp dessa och tryck på mellanslag. Nu kan vi skriva något som kallas för template string inom dessa backticks. Översta raden bör alltså se ut som i bild 17 innan vi har lagt dit något innehåll.

```
kaninData.forEach((kanin) => {
  kaninLista.innerHTML += ``
});
```

Bild 17 For-loopen innan vi har lagt till innehåll, alltså bara += och backticks.

Nu kan vi gå till html-dokumentet och kopiera ett helt list-element med klasser, text och hela hootabatjooet, enligt bild 18.

```
<li class="card">
  <h3 class="card-namn">Nisse Snubbelfot</h3>
  <div class="card-beskrivning">
    <p>Nisse är glad men något klumpig.</p>
  </div>
</li>
```

Bild 18 Hela hootabatjooet.

Allting ska nu klistras in mellan de båda backtick-apostroferna. Men vi vill ju inte att varje post heter Nisse Snubbelfot och har hans beskrivning, utan vi vill ju hämta den från varje kanin i listan. Därför byter vi ut namnet till variabeln kanin.name. Om man nu använder det som kallas för template strings så kan man skriva in variabeln med ${kanin.name} . Då kommer hela funktionen skapaKaniner att se ut som på bild 19.

```
function skapaKaniner(kaninData) {
  kaninRubrik.style.display = "block"
  kaninLista.style.display = "grid"

  kaninLista.innerHTML = ""

  kaninData.forEach((kanin) => {
    kaninLista.innerHTML += `<li class="card">
       <h3 class="card-namn">${kanin.name}</h3>
       <div class="card-beskrivning">
         <p>${kanin.company.catchPhrase}</p>
       </div>
     </li>`
  })
}
```

Bild 18 Hela hootabatjooet.

Observera att vi rensar bort Kalle Kanin och Nisse Snubbelfot på raden ovanför for-loopen för de är ju inte ”riktiga” kaniner som vi har hämtat utan påhittade. Om allt är rätt så ska det hända lite saker när vi trycker på knappen kaniner nu. Öppna gärna developer tools med Console för att kolla eventuella felmeddelanden om inte allt fungerar.