Individuellt projekt #2

FE19 KYH Stockholm

Kim Nkoubou, VT2020

Innehåll

Inledning

Den praktiska delen i uppgiften har gått ut på att skapa en portfolio-sida med library:t React. Sidan

- ska vara gjord utan hjälp av någon hemsidegenerator eller bootstrap/react-strap och liknande.
- ska nyttja minst ett React-bibliotek utöver React och React-DOM.
- ska hämta extern data från valfrit API med hjälp av AJAX.
- får nyttja vanilla CSS, SASS, LESS eller Styled Components
- ska möjliggöra kontakt inklusive Github-länk.

Denna redogörelse tar upp det viktigaste kring ett antal relevanta frontend-punkter och knyter detta till den praktiska delen.

CSS-ramverk

CSS-ramverk är bibliotek som kan underlätta designen av UI och dess komponenter. I praktiken skulle man kunna se CSS-ramverk som ett slags tema innehållandes färdigdesignade komponent-mallar redo att distribueras i ett webbprojekt. Vissa ramverk använder sig enbart utav CSS-syntax men några tar steget längre med Javascript-funktionalitet. Utöver Bootstrap är Foundation, Materialize och Semantic UI tre utav en rad populära sådana.

Semantic UI

Detta ramverk fokuserar på att göra designprocessen mer semantisk och koden lättare att läsa och förstå. Det nyttjar ett naturligt språk där klasser behandlas som editerbara komponenter. Semantic UI har en stor mängd inbyggda theme-variabler och unika komponenter, omfattande dokumentation och ett inspirerande antal Github-commits.

Foundation

Foundation är det första, och enligt dess skapare, världens mest avancerade responsiva front end-ramverk. Det är responsivt, semantiskt, har ett mobile first-tänk och ett flertal unika UI-komponenter. Foundation är open source. Det preprocessas med SASS och är kompatibelt med i stort sett alla webbmiljöer som levererar HTML (ex: Wordpress, .NET). Av alla stora CSS-ramverk är Foundation det enda som upprätthålls och utvecklas av en professionell organisation (ZURB). Dokumentationen är omfattande inklusive, övningstutorials, support och en hel del externt resursmaterial.

Materialize

Vad gäller Googles material design är Materialize ett av de mest kända CSS-ramverken. Och för en som själv studerar är det inspirerande att Materialize skapats av just studenter. Det preprocessas med SASS, är responsivt och har

unika komponenter. Ramverket är en mycket bra ingång till att jobba med material design då det, trots sina customization-möjligheter, följer material designs principer. Dokumentationen är bra och det finns en support- och diskussionsplatform i chattformat.

Val av ramverk

När man ska välja ramverk kan det vara svårt att veta vilket som lämpar sig bäst för ett projekt. Att lista nödvändig funktionalitet och resurser för att sedan jämföra med vad utbudet erbjuder är bra. Men det är inte alltid en vet exakt vad projektet behöver, särskilt inte om ramverken en sneglar på är okända för en. Att gå efter några generella parametrar kan då optimera chanserna att välja rätt:

- Stor user-base: visar på att ramverkets testats utförligt och har hög kompatibilitet med tredjeparts-extensions.
- Aktiv contributor-base (många pull-requests): visar på att ramverket lever, utvecklas och uppdateras.
- Täta releaser: visar på att utvecklingen går fort och att en inte behöver vänta alltför länge på uppdateringar.
- Utförlig dokumentation: minskar risken för att fastna i problem relaterade till syntax och kod i allmänhet. Om ramverket är nytt för utvecklaren innebär utförlig dokumentation också att det innehåller avsnitt som är anpassade för nybörjare.
- Snäv inlärningskurva: ökar chanserna att komma igång snabbt. Om ramverket kräver stor förkunskap eller särskilda teknologiska förutsättningar påverkar detta.

Semantic UI, Foundation och Materialize ligger samtliga relativt bra till på dessa punkter.

CSS-ramverk i portföljen

På sidan har jag valt att inte nyttjat Bootstrap eller liknande på grund av spec-kraven. På sikt kanske jag implementerar något för att prova - jag tycker att material design känns intressant, men såsom designen ter sig just nu skulle jag få tänka om lite.

CSS-extensions (preprocessors)

En CSS-preprocessor är en CSS-extension som läser sitt (oftast objektorienterade) språk/syntax och omvandlar/kompilerar koden till vanlig hederlig CSS.

Fördelar

Den största mest självklara fördelen med att använda en preprocessor för CSS är att, i takt med att ett projekt växer, koden blir lättare att hålla torr / icke-redundant. De största CSS-preprocessorerna är SASS och LESS och tillhandahåller en rad behjälplig funktionalitet som CSS saknar:

- Variabeldeklaration/uttryck
- Operationer/beräkningar
- Mixing: funktioner för att tillskriva selector/variabel-grupper värden/properties och även att omvänt som en slags theeming)

- Nestade block, likt html-hierarki
- Extend: tilldela en selector properties från en annan
- Importerar multipla filer med en enda HTTP-request

I nuläget har SASS flest användare, men båda språken har ett snarlika funktionella möjligheter. SASS har två olika syntax-versioner: SASS.sass och Sassy-CSS.scss (med respektive filtyper). Båda använder SASS preprocessor men har en markant syntax-skillnad. Kodblocken och hierarkin i SASS utgörs av intendering medan SCSS likt CSS jobbar med {måsvingar/curly brackets}. De flesta SASS:are använder SCSS, och en fördel är att SCSS i praktiken är bakåtkompatibelt (på så sätt att man kan importera vanlig CSS-kod i en scss-fil).

Kod exempel

CSS.css

```
.parentClass {
    padding: 77px;
    font: 100% Helvetica, sans-serif;
    color: #333;
}
.childClass {
    padding: 37px;
    font: 100% Helvetica, sans-serif;
    color: #333
```

SASSY.scss

```
$paddingLarge: 77px;
$paddingSmall: $paddingLarge - 40px;
$font: 100% Helvetica, sans-serif;
$color: #333;

.parentClass {
    padding: $paddingLarge;
    font: $font;
    Color: $color;

            .childClass {
                padding: $paddingSmall;
                font:$font;
                color: $color;
            }
}
```

LESS

```
@paddingLarge: 77px;
@paddingSmall: @paddingLarge - 40px;
@font: 100% Helvetica, sans-serif;
@color: #333;

.parentClass {
   padding: $paddingLarge;
   font: $font;
   Color: $color;

          .childClass {
             padding: $paddingSmall;
             font:$font;
             color: $color;
          }
}
```

CSS-extension i portföljen

På sidan används en mix utav Styled Components (injicerar CSS direkt i React-komponenter), SCSS och CSS. Anledningen till att jag har mixat är att vanlig klass-hänvisning/tillskrivning fungerat smidigare med vissa bibliotek som använts (exempelvis CSS-transition). Förmodligen har det med min brist på expertis att göra och jag behöver utforska i vilken grad jag kan vara mer konsekvent. Eftersom sidan är ett relativt litet projekt känns det dock okej att mixa lite, men hoppas på framtida refaktorering i takt med att min React-kunskap ökar.

Javascript - syntax

Kodexempel - grundläggande syntax

Följande kodsnutt består i en variabeldeklaration (msg) samt en eventhandler-funktion.
Justering av webbläsarens fönsterstorlek triggar funktionen som i sin tur kör en alert-funktion med den tidigare deklarerade variabeln som argument. Det hela resulterar i att kod-exekveringen pausas och en dialogruta med texten “Ahaa!” visas i webbläsaren.

```
var msg = 'Ahaa!';

window.onresize = function(){
            alert(msg);
});
```

Kodexempel från portföljen

Denna kodsnutt är en vidareutveckling (och modernisering => ES6-syntax) av den förra. Till skillnad från förut är nu eventhandlern inkapslad i en yttre funktion och det händer en massa annat trevligt. Syftet med kommande kodsnutt är att visa ett meddelande först när användaren är klar med att justera fönsterstorleken. Det finns inget inbyggt sätt för att åstadkomma detta (borde förmodligen ha googlat bättre), därav funktionen afterResize. När vi kallar på funktionen från sista raden sker hur som helst följande:

1. Argumenten vi skickade med (en sträng och ett tal) representeras inom funktionen av variablerna callback och timeout.
2. Vi deklarerar den editerbara variabeln doWhenDone men tillskriver den inget värde (undefined). Mystiken kring varför vi gör detta avtar längre fram.
3. Fönster-objektets property onresize var förut lika med null. Nu tillskriver vi den, liksom i förra kodsnutten, värdet av en funktion. Om en i detta läge skulle consolla window.onresize skulle en få tillbaka själva funktionsdeklarationen. För att funktionen ska köras behövs alltså en riktig trigger. A.k.a. En faktisk ändring av fönstrets storlek. Och då sker följande:
    - 3.1. Vi har tillgång till den tidigare deklarerade variabel doWhenDone pga javascripts closure-arkitektur.
    - 3.2. Vi kör web API-funktionen clearTimout med den tidigare deklarerade variabeln doWhenDone. Första gången denna eventhandler körs är doWhenDone dock undefined så inget särskilt sker (mystiken kommer avta).
    - 3.3. Efter detta kommer den stora/lilla twisten: Nu får doWhenDone plötsligt värdet av web-API-funktionen setTimeout (vilkens innehåll körs efter att eventloopen fångat in den web-API:n):
        - 3.3.1. setTimeout kör i sin tur afterResize:s första parameter: callback (vilket är alert('Ahaa!'). MEN detta sker först efter afterResize:s andra parameter-värde, timeout, i millisekunder. Och eftersom afterResize:s andra parameters defaultvärde skrevs över innebär detta att alert('Ahaa!') alltså körs om en halv sekund.

```
const afterResize = ( callback, timeout = 300 ) => {
  let doWhenDone;

  window.onresize = () => {
    clearTimeout(doWhenDone);

    doWhenDone = setTimeout(() => {
                callback();
    }, timeout);
  };
}
afterResize( alert('Ahaa!'), 500 );
```

Recap:

- Om eventhandlern inuti afterResize bara skulle köras en gång (inte troligt) visas en dialogruta efter 500ms / en halv sekund och clearTimeout har då ingen riktig poäng.

- Om användaren däremot fortsätter justera fönsterstorleken (troligare) avbryts den nyss startade och nedtickande setTimeout (vid namn doWhenDone) som planerar att köra callback mindre än en halv sekund. Det är i detta scenario som clearTimeout alltså fyller sitt syfte.

- På detta sätt planerar vi att visa dialogrutan en massa gånger (varje gång eventhandlern triggas) men gör det bara om fönsterstorleken inte justerats på en halv sekund, vilken med relativt stor säkerhet påvisar att användaren slutat justera fönsterstorleken.

Syntax i portföljen

En variant av den sista kodsnutten används i skrivande stund på sidan men löper dock risk för att ganska snart antingen försvinna, skrivas om eller ersättas eftersom det från början var lite av en nödlösande workaround. Men fram till att man blivit en React-jävel är det kanske ingen idé att vara finsmakare.

Javascript - versioner och kompatibilitet

Javascript skapades av Brendan Eich 1995 och bygger sedan 1997 på den officiella språkstandarden ECMAscript. 2000-2010 refererade Mozilla explicit till aktuell javascript-version (de tre sista var 1.8, 1.82 respektive 1.85). När det numera talas om javascript-versioner syftas det (om jag inte missförstått allt och får gå om en klass på KYH) snarare aktuell ECMAscript-version och vilka nya javascript-features som stöds. Sedan 2015 (ECMAscript version 6 eller ES6) har det släppts nya ECMAscript-versioner årligen. Den allra senaste sjösattes/godkändes juni 2019 och de tre senaste - om vi bortser från ESNext (nästa/nuvarande version under konstruktion) benämns således som ECMAscript 8, 9 respektive 10 (eller ES8-10). Varje version har kommit med nya inbyggda features och det som versionerna 4+ har gemensamt är att de i sig själva inte är kompatibla med samtliga webbläsare utan i vissa fall behöver en (compiler/)transpiler (exempelvis Babel) som omvandlar modern syntax till javascript-syntax av äldre ECMA-standard. Alla moderna webbläsare stödjer däremot ES5 fullkomligt.
ES7-10 (2016+) har enligt Kangax (https://kangax.github.io/compat-table/es2016plus/) alla samma procentuella kompatibilitet med den modernaste versionen av följande webbtjänster:

- Babel 7 + Core.js 3: 65%
- Typescript + Core.js 3: 62%
- Internet Explorer: 0%
- Firefox 68-74: 87%
- Chrome 78-81: 100%
- Opera 65-68: 100%
- Edge: 92%

- Safari: 88%
- Node: 92%
- iOS: 80%
- Samsung Internet: 82%
- Opera Mobile: 92%

Version i portföljen

När det kommer till rena JS-block har jag i skrivande stund i stort sett inte nyttjat något utöver ES6. Däremot funderar jag i min hybris lite löst att titta på modernare features såsom async/await (ES8) och om "rest-props for object destructuring" (ES9) är värt att utforska för ett globalt states reducer-funktion lite längre fram.

Javascript - exekvering i webbläsaren

Det finns ett flertal exekveringsmiljöer för javascript men den i särklass vanligaste är webbläsaren. En HTML-sida från servern laddas in och tillhörande javascript-kod körs. De kugghjul som samverkar i processen att exekvera javascript i webbläsaren - bortsett från själva koden/scriptet - är webbläsarens Web API:er (Browser Object Model/BOM API:s) och en javascript-motor, vilka samverkar enligt en event loop-modell.

Web API:er

Bland webbläsarens Web API:er ingår bl.a. DOM API:n som möjliggör interaktion mellan javascript och sidans nodstruktur samt funktioner som setTimeout och setInterval. Javascript använder sig utav Web API:ernas resurser men är i sig självt inte en del utav dem. När en javascript-kod kallar på en API-funktion körs denna (med ett par blockande undantag) parallellt med att exekveras koden vidare.

Javscript-motor

Javascript-motorns uppgift är att översätta/bearbeta/optimera och köra koden. Precis som det finns olika webbläsare finns det olika motorer:

- SpiderMonkey
    - den första javascript-motorn
    - utvecklad av Mozilla
    - körs av Firefox
- Nitro
    - körs av WebKitprodukter som Safari
    - utvecklad av Apple
- Chackra
    - körs av Edge

    - Utvecklad av Microsoft
- V8
    - körs av bl a Chrome och Opera
    - utvecklad av Google

är några utav de största.

Javascript-motorn har två viktiga enheter: Call Stack och Memory Heap.

Call Stack

Javascript utför ett enda call/process/uppgift i taget (javascript är single threaded) och Call Stacken visar vilken process/uppgift som för tillfället exekveras. En efter en åker uppgifterna in och ut ur Call Stacken. Så fort en uppgift kallar på en annan uppgift och vi hoppar ett steg djupare i exekveringskontexten stannar däremot den "yttre huvuduppgiften" kvar och den "inre underuppgiften" lägger sig ovanpå. När en "yttre huvuduppgifts" alla "underuppgifter" är klara är också den "yttre huvuduppfiten" klar och lämnar plats åt nästa uppgift i likvärdig exekveringskontext. Denna ordning/struktur kan benämnas som "sist in, först ut". Exempel:

1. Javascript-kod börjar köras. Vi befinner oss i global scope och Call Stacken har en global exekveringskontext.

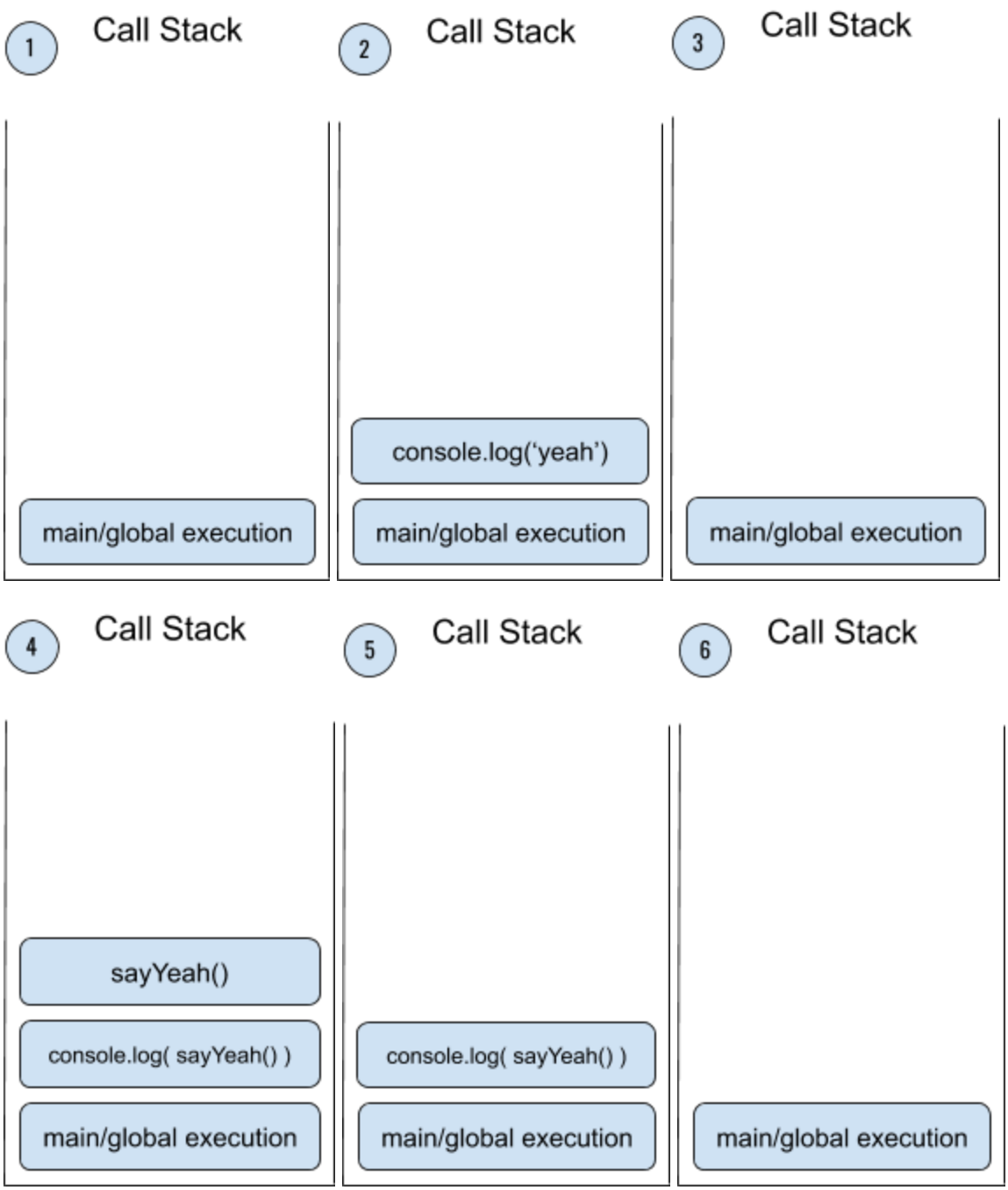
2. Plötsligt dyker det upp en rad i koden som ser ut så här: **console.log('yeah!');**
   I vår globala kontext kallar vi alltså på funktionen console.log och Call Stacken byggs på.

3. Nu är uppgiften console.log('yeah') klar och hoppar således ut.

4. Nästa rad ser ut så här: **console.log(sayYeah());**
   (Funktionen sayYeah returnerar strängen 'yeah!').
   Vi kallar på uppgiften console.log() som i sin tur kallar på sayYeah

5. sayYeah returnerar nu strängen 'yeah!' och är klar.

6. Consolen loggar 'yeah!' och vi är tillbaka där vi var, i vår globala exekveringskontext.



Recap:

- Alla likvärdiga uppgifter (kontextmässigt) körs rad för rad, åker in och ut ur Call Stacken en efter en.
- Exekveringen äntrar en djupare kontext: nytt Call Stack block lägger sig ovanpå.
- Exekveringen lämnar den djupare kontexten: översta blocket hoppar bort.

Event Loop

Event-loopens uppgift är att samla in och bearbeta event och köra (skicka till Call Stack) köande processer (messages in a message queue). Processerna körs en i taget, med turordningen äldst först. När en funktion som setTimeout tickat ner hamnar den i kön och får i bästa fall köras direkt.

Memory Heap

TIll skillnad från low level-språk sköts minnsehantering och garbage collection automatiskt i javascript. Men sättet moderna javascriptmotorer som V8 jobbar på gör att kodens uppbyggnad kan vara avgörande för hur optimerad allokering och garbage collection blir. V8, en av de mest populära javascript-motorerna, använder något som heter hidden classes, vilka deklarerade objekt pekar mot, för att lägga data på minnet.

Transpiler vs compiler

De stora moderna webbläsarna använder en JIT compiler (Just In Time) för att just före exekveringen översätta javascript till körbart byte-format. V8, som byggdes för att effektivisera processen, hoppar däremot över detta steg och översätter javascript till optimerad maskinkod direkt. V8 har till skillnad från javascript flera trådar och i denna runtime-process nyttjas de parallellt:

- Kompilerar all javascript till maskinkod för snabb start av exekvering
- Under exekvering analyseras koden optimeras/refaktoreras för snabbare fortsatt exekvering. Detta innefattar bl.a. att funktions-anrop ersätts med funktions-kroppar och att memory-lookups ersätts med direkta värden.
- Under exekveringen byts originalkoden ut mot den optimerade versionen.
- Garbage collection utförs i uppdelade sjok, vilket skapar fler men kortare pauser i exekveringen.

Exekvering av portföljen

Att välja motor är såklart lite svårt då det beror på vilken webbläsare som skickar HTTP-requestet. Däremot har jag, med tanke på hur javascript körs och hur minnesallokeringen fungerar, funderat en hel del på hur jag kan underlätta för motorn i min sidans kod. Jag kan inte påstå att jag hittills lyckats jättebra. Den mesta energin har av naturliga skäl lagts på att bara få saker att funka. Optimerad kod har kommit i efterhand. I skrivande stund tänker jag särskilt på strategin i den globala reducer-fuctionen (StateContext) där jag vid varenda uppdatering kopierar objekt-properties med hjälp av en loop. Vidare efterforskning kring om object.assign eller liknande är bättre krävs!

Javascript - bibliotek

Ett javascript-bibliotek är förskriven javascript-kod som kan underlätta implementeringen av UI eller integrationen mellan olika webbteknologier och scriptspråk. Det finns en väldigt stor mängd tillgängliga js-bibliotek och några riktigt populära är

- Parsley: verktyg för att validera data, exempelvis från ett formulär

- D3.js: UI-verktyg som underlättar visualiseringen av data
- jQuery: möjliggör förenklad javascript-syntax och DOM-manipulation.

Bibliotek i portföljen

React i sig självt är ett bibliotek (ibland benämnt som ramverk) och omsluter det mesta av sidans innehåll och funktionalitet. Reacts uppgift är att underlätta skapandet av UI:s och kan ses som view-delen i en Model View Controller-kontext. Den är komponentbaserad och nyttjar JSX (ett script-språk som ser ut som HTML och javascripts oäkting). Processen med att använda javascript-biblioteket React har hittills inkluderat mycket glädje och ungefär lika mycket frustration då React har en alldeles särskild relation till webbläsarens DOM-modellen.

Några bibliotek utöver React från NPM Registry som jag funnit användbara för sidan är:

- Styled Components: möjliggör komponentbaserad (tar in props), preprocessad css-styling formulerad som javascript-syntax i string-literals.
- CSS-Transitions (React Transition Group): applicerar särskilda klasser på komponenter utifrån render-status och möjliggör, via vanlig css, således transition-animationer.
- Fort(font)awesome: icon-bibliotek

I detta, mitt första React-projekt upplever jag det som väldigt lätt att hamna i “WP-plugin-fällan”. Även om det finns tusentals javascript-bibliotek tillgänglig för vilken vanlig HTML-sida som helst känns det, just pga tillgången till NPM, upplagt för att en ska hitta de bästa, mest kvalitativa biblioteken som kommer “rädda appen”. En reflektion jag får utvärdera på nytt längre fram.

Javascript - ramverk

Ibland är det inte helt lätt att särskilja ett bibliotek från ett ramverk. En förenkling skulle kunna vara att ett bibliotek bara är en signerad låda med funktioner som du kan kalla på. Ett ramverk skulle då i sin tur vara en mer omfattande och komplex arkitektur, i vilken du behöver implementera din kod utefter ramverkets design och principer - ärva/förgrena ramverkets klasser eller möjligen injicera dina egna. Och till skillnad från biblioteket är det ramverket som kallar på din kod. Men få beskrivningar är vattentäta hela vägen.

Vissa kallar React för ett ramverk. De flesta verkar dock eniga i att ett ramverk bör stå för fler bokstäver än V i MVC-strukturen. Det är dessutom ganska enkelt att skapa en, på ytan, React-app där 99,9% av koden inte har någonting med React att göra.

Några av de populäraste ramverken idag är Angular, Ember, Vue och Svelte (på uppgång?). Dessa ramverk åstadkommer enligt briefa beskrivningar i stort sett samma saker: moderna dynamiska webb-applikationer och UI:s. Ember och Vue (och förmodligen Svelte) bygger däremot på Model-View-Viewmodel-arkitekturen (MVVM) medan Angular har en Model-View-Whatever-variant (MVW). Och till skillnad från de andra använder Angular 2+ ECMA-språket Typescript.

Pros med att använda ramverk

- Erbjuder en redan utformad app-arkitektur/kod-struktur, vilka kan hjälpa till att behålla strukturen i takt med att appen växer.
- Inkluderar vanligtvis en mängd features/funktionella genvägar, vilket kan minska mängden kod som behöver tillföras för att åstadkomma något, och på så sätt effektivisera arbetet.
- Ofta har ramverket redan löst potentiella problem, såsom webbläsarkompatibilitet och säkerhetsaspekter.
- Backas ofta upp av både utvecklarteam och stora communities som bidrar till utveckling, test och support. Detta kan förenkla säkerställandet av programmets säkerhet och kvalitet.
- De flesta ramverk är open source, vilket, om ramverkets fördelar visar effekt, kan ha en ekonomisk fördel för ett projekt.

Pro + Con

- Beroende på ramverk och tidigare erfarenhet kan det vara mycket nytt att sätta sig in i (ex syntax, arkitektur) före byggprocessen kan dra igång, dvs potentiellt planare inlärningskurva. Men det kan vara precis tvärtom också. Ramverket kan vara lättskrivet, ha utmärkt dokumentation för nybörjare och massor av pedagogiska användare att vända sig till i forum.

Cons med att använda ramverk

- I ett litet projekt kanske ett ramverks alla features blir överflödiga. I många fall jobbar Vanilla (vanlig/plain) javascript allra snabbast, och då kan implementationen av ett ramverks hela kodbas vara en potentiell omväg och ett sämre alternativ.
- Med Vanilla javascript är koden helt oberoende av ett ramverks uppdateringar.

Ramverk i portföljen

Det finns säkert många lyckade exempel på implementationer av både React och ett ramverk i samma webb-app. Men då detta är ett React-projekt har React fått monopol.

DOM:en

“Den reella DOM:en” är inget vedertaget begrepp, men används här för att skilja DOM:en från den virtuella DOM:en.

Den reella

Vi börjar med att zooma ut några steg. Webbläsaren är ett program vars huvuduppgift brukar vara att läsa HTML, CSS och script på webbsidor. När ett webbläsarfönster öppnas instansieras ett window object, vilken representerar det aktuella fönstret/tabben och vars properties och metoder kan nås med hjälp av ett programspråk som kan interagera med webbläsaren, exempelvis javascript. Window object innehåller i sin tur bl.a. document object vilken representerar sidans innehåll (HTML/CSS) utifrån källkoden. I Chromes devtools (eller liknande) kan man jämföra vad som visas under elements-tabben med vad window.document ger tillbaka i konsolen. Innehållet är identiskt. DOM:en utgörs av document object och de metoder som ärvts av window object. En snarlik definition skulle vara att DOM:en består av ett objekt avbildat utifrån källkodens HTML samt ett interface (webbläsar-API) för att manipulera detta objekt. DOM-objektet har en trädstruktur och innehåller/förgrenas ut i objekt som representerar elementen på sidan. Dessa objekt kallas för noder och ur ett objekt-perspektiv (inte i sig självt applicerbar syntax) skulle förhållandet mellan document object och ett division-element kunna se ut så här: document.body.div.

Utav källkoden framställer alltså webbläsaren en slags avbild (DOM:en) i objektformat med noder som properties och alla tillhörande property-värden och metoder. DOM:en visas för användaren, och när vi manipulerar DOM:en med javascript uppdateras det som visas. Däremot ändras då inte källkoden eftersom DOM:en lever i webbläsaren på klientsidan, och källkoden lever på servern. Webbläsaren kan även optimera/rätta till fel funna i källkoden innan DOM:en renderas. Alla dessa förutsättningar gör att källkoden och DOM:en ofta inte är identiska.

Den virtuella

Den virtuella DOM:en är en kopia av den reella i form av ett objekt. Ramverk/bibliotek som React/Vue använder den virtuella DOM:en, och när den manipuleras via ramverket sker ungefär följande i mycket förenklade drag:

1. Den reella rörs till en början inte.
2. Det skapas en uppdaterad kopia av det virtuella och skillnaderna mellan kopian och det virtuella originalet samlas in till något som kallas diffs.
3. Den virtuellas API optimerar instruktioner utifrån alla diffs och ber den reellas API att uppdatera den reella på ett effektivt sätt.

Att manipulera den riktiga DOM:en via javascript tenderar att bli omständigt när ändringarna är omfattande och frekventa. Den främsta fördelen med att nyttja den virtuella och dess API är att det förenklar interaktionen med den reella samt optimerar/effektiviserar de instruktioner som utför ändringarna. React har sålt in den virtuella DOM:en p.g.a. sin snabbhet. Samtidigt är följande citat, uppbackat av många som inte förespråkar den virtuella DOM:en, svårt att argumentera emot:

"Virtual DOM is, by definition, slower than carefully crafted manual updates, but it gives us a much more convenient API for creating UI." - *https://medium.com/@hayavuk/why-virtual-dom-is-slower-2d9b964b4c9e#1bac*

AJAX

Asynchronous Javascript And XML är en samling webbteknologier som bl.a. gör det möjligt att läsa från/skriva till en server och uppdatera delar av en webbsida, och således inte behöva ladda om hela sidan. Att namnet innehåller XML kan vara lite missvisande då det förmodligen är lika vanligt att transportera data i json-format.

Några fördelar:

- Att kunna uppdatera en sida utan att ladda om den ger fler “checkpoints” under processen att validera och bearbeta data. Exempelvis kan input i ett formulär valideras och “submitas” under tiden det fylls i.
- Att endast hämta specifik ny data från servern och behålla HTML:n hos klienten tar generellt kortare tid och nyttjar mindre bandbredd.
- Det kan underlätta särskiljandet mellan HTML/layout och data i en webb-applikation.

Ett par nackdelar:

- I vissa fall kan historik-navigation bli problematiskt.
- Att bokmärka en sidas specifika status är inte alltid möjligt.

AJAX i portföljen

På /music-sidan hämtas material från Spotify. Detta är med hjälp av ett Iframe-element med extern source och således requestas HTML utan javascripts hjälp. I skrivande stund inser jag att jag glömt bort att detta inte uppfyller spec-kravet: hämta extern data med hjälp av AJAX. Så här några timmar före deadline får vi se om jag kan lyckas lösa fadäsen.

Edit: Nu så här på inlämningsdagen har jag slängt upp en temporär subsida (/ajax) där det hämtas lite fake-users. Dock har jag valt att använda Axios, ett smidigt library för att göra det uppgiften efterfrågar.

HTTP

Hypertext Transfer Protocol ett protokoll som ingår i det översta lagret (applikations-lagret) i arkitekturen för datakommunikation över nätverk (TCP/IP). HTTP-standarden skapades 1989 och bygger på att klienten sänder ett textbaserat meddelande till en server genom HTTP-metoden GET eller POST och får tillbaka ett svar, i webbläsarkontext vanligen i form av en HTML-sida. De stora uppdateringarna sedan den första dokumenterade versionen 0.9 (1991) har varit HTTP 1.0 (1996), 1.1 (1997) respektive HTTP 2.0 (2015) och 3.0 (2018).

1.1 vs 2.0

När 1.1 kom blev den snabbt standarden för internet. En utav flera stora uppgraderingar från 1.0 var att en request-session mellan klienten och servern kunde hållas öppen fram till att den aktivt stängdes (keepalive-connection). Ett långdraget scenario klienten då äntligen slapp var att skicka upprepade requests ett efter ett till servern fram tills att hela sidan med samtliga resurser laddats ner.

2.0 utvecklades av Google för att snabba på leveransen av webbsidor. 2.0 stöds idag av i princip alla stora webbläsare och webbservrar och versionen är dessutom bakåt-kompatibel med föregångaren. Några punkter där 2.0 skiljer sig från 1.1:

- Istället för rent textformat transporteras meddelanden i binärt (ettor, nollor) format vilket är mer kompakt och effektiviserar transporten.
- De binära meddelanden kan delas upp i mindre delar och transporteras parallellt och flexibelt - oberoende av ordning under transport för att samlas upp/sorteras först vid ankomst (2.0 tillåter servern att ge webbläsaren prio-ordning) - vilket minimerar risken för att ett meddelande blockerar ett annat. Detta kan dessutom ske via en enda TCP-anslutning.
- Uppdelningen av header och body (som paketeras om med nya “titlar/rubriker”) möjliggör kompression, vilket också effektiviserar transporten.
- Med färre error-helpers är 2.0 mindre benägen att producera “onödiga” fel. Och onödiga fel är onödiga avbrott.
- Alla textuella intrångsmöjligheter försvinner med det binära formatet.
- Med server push levereras HTML och nödvändiga resurser vid första svaret från servern istället för en resurs i taget.
- 2.0 nyttjar caching-metoder för resurser för att ladda in dem snabbare på andra sidor.

HTTP för portföljen

Sidans besökare kommer nog uteslutande att nyttja protokoll-version 2.0. I skrivande stund är jag osäker på webbhotellets kompatibilitet, men givetvis ska jag så fort som möjligt - bara för att - ladda ner Chrome Canary och aktivera HTTP 3.0!

Min portfölj

Detta projekt har varit ett jättebra sätt att kasta sig in i UI-biblioteket React. Det har gått fort, och mycket har säkert gått ut genom andra örat, men att få jobba med Node, NPM och komponentbaserat har givit otroligt mycket nya insikter och inspiration till framtiden!

Det räcker att snegla på källkoden på github för att förstå att sidan är under konstruktion och kräver allmän uppstädning (public-mappen är exempelvis inte jätterolig). Nuvarande version får dock ligga uppe hos Surge, och när jag är någorlunda nöjd ska den få ersätta https://kiiim.se. Det härliga med att uppgiften är ett verkligt portfolio är att det känns helt naturligt att den ännu inte är klar. Den får vara levande, utvecklas, uppdateras och refaktoreras i en tid framöver, i takt med att jag lär mig React och reder ut vad jag har för layout-smak. Over n out!

http://kimnkoubou-portfolio.surge.sh

Källor

- https://www.thewebmaster.com/hosting/2015/dec/14/what-is-http2-and-how-does-it-compare-to-http1-1/
- https://zurb.com/
- https://en.wikipedia.org/wiki/Material_Design
- https://materializecss.com/
- https://www.keycdn.com/blog/sass-vs-less/
- https://en.wikipedia.org/wiki/JavaScript#Version_history
- https://babeljs.io/
- https://kangax.github.io/compat-table/es2016plus/
- https://www.freecodecamp.org/news/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management