ARCADA

Funktionell reaktiv programmering i användar-gränssnitt

Niclas Blomberg

Examensarbete
Informations- och medieteknik
2016

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	
Författare:	Niclas Blomberg
Arbetets namn:	Funktionell reaktiv programmering i användargränssnitt
Handledare (Arcada):	Johnny Biström
Uppdragsgivare:	

Sammandrag:
Detta arbete behandlar funktionell reaktiv programmering i användargränssnitt. Arbetet fokuserar på webb baserade teknologier, men principerna gäller även i andra omgivningar. Arbetet kommer först att beskriva de teknologiska framsteg som gjort FRP möjligt på webben och kommer sedan att jämföra FRP med imperativ DOM manipulering. Huvudsakliga målet är att försöka motivera varför FRP med en arkitektur som bygger på enkelriktat dataflöde är ett effektivt sätt att bygga användargränssnitt.

Eftersom arbetet bygger delvis på kod exempel, måste vissa JavaScript bibliotek behandlas kort. Poängen är dock att behandla FRP som en helhet och inte gå in på djupet med specifika implementationer.

Nyckelord:	Funktionell reaktiv programmering, JavaScript
Sidantal:	
Språk:	Svenska
Datum för godkännande:	

DEGREE THESIS	
Arcada	
Degree Programme:	Information- and media technology
Identification number:	
Author:	Niclas Blomberg
Title:	Functional reactive user interface programming
Supervisor (Arcada):	Johnny Biström
Commissioned by:	

Abstract:
This thesis takes a look at how functional reactive programming can be used to create user interfaces. The implementations discussed are web-based, but the theory behind it is applicable to other environments as well. The thesis starts by presenting the problem with imperative view code, then it takes a look at new technologies that make reactive user interfaces possible on the web. Lastly an architecture pattern using these technologies is presented, and arguments for it's effectiveness are made.

Because this thesis builds on a lot of code examples some JavaScript libraries are covered. The main point is not to inspect these libraries in depth, though, but to take a look at the principles behind functional reactive user interface programming.

Keywords:	Functional reactive programming, JavaScript
Number of pages:	
Language:	Swedish
Date of acceptance:	

INNEHÅLL / CONTENTS

Figurer

1 INLEDNING

Webben är inte längre endast ett nätverk av statiska dokument utan den har utvecklats till en egen applikationsplatform. Webbens historia som en plattform för statiska dokument har hämtat med sig ballast och försvårat modern applikationsutveckling på webben. Under åren har flera olika programmeringsparadigmer och designmönster prövats på webben, men eftersom DOM modellen inte ursprungligen är gjord för dynamiskt innehåll och avancerade användargränssnitt har ingen av dem skapat en långlivad trend.

Asynkron programmering blir lätt svårt eftersom det snabbt uppstår komplicerade beroendesamband mellan olika komponenter och delar av koden. Eftersom användargränssnitt måste ta emot inmatning av människor är användargränssnitt i grund och botten asynkrona.

1.1 Syfte och mål

Syftet med arbetet är att undersöka vad FRP är, dess nyttor och hur det kan implementeras på webben. Sedan kommer arbetet att gå in på hur man kan använda FRP för att skapa ett effektivt ramverk för webbapplikationer.

Målet är att ge läsaren en klar bild av FRP och Flux-inspirerade designmönstret som bygger på enkelriktat dataflöde, samt motivera varför detta är ett produktivt sätt att utveckla användargränssnitt på webben.

1.2 Metodik

Jag har under senaste året varit med och jobbat på en avancerad webbapplikation som är byggd på dessa principer och teknologier. Arbetet bygger på mina erfarenheter från detta projekt och argumenten kommer att stödas m.h.a. kod exempel, teknisk dokumentation och vetenskaplig litteratur. Eftersom koden till webbapplikationen ägs av ett företag kommer jag inte att behandla dess detaljer utan istället koncentrera mig på de utnyttjade principerna och teknologierna.

1.3 Avgränsning

Eftersom arbetet kommer att innehålla kod exempel kommer jag att använda mig av vissa JavaScript bibliotek i exemplen. Vissa teknologier och delar av biblioteken kommer att behandlas så att läsaren kan förstå kod exemplen, men fokusen ligger inte på enskilda biblioteken. De flesta av de behandlade biblioteken har konkurrenter och kunde lätt ersättas med någon av dem.

Huvudvikten i arbetet kommer att ligga på att markera problemen och svårigheterna i utveckling av användargränssnitt på webben samt beskriva hur FRP kan hjälpa lösa dessa.

2 DOCUMENT OBJECT MODEL (DOM)

DOM (Document Object Model) är ett gränssnitt som möjliggör interaktion med objekt i XML och HTML dokument (Document Object Model 2016). Webbläsarna använder DOM som gränssnitt mellan webbläsarens inre representation av en nätsida och JavaScript. Via DOM gränssnittet kan JavaScript kod läsa och manipulera strukturen av DOM trädet, vilket möjliggör utveckling av dynamiska webbsidor. Ett vanligt mönster är att först återge en representation av vyn i en applikation i DOM, och sedan manipulera strukturen i respons till användarens interaktion (visa och gömma element, ändra textinnehåll, skapa nya element osv). Biblioteket jQuery underlättar denna manipulation, men principen är ändå den samma (jQuery 2016).

2.1 Problem med DOM

Det finns vissa problem med denna modell då det kommer till mer avancerade gränssnitt. Moderna JavaScript motorer som V8 och Chakra har utvecklats till mycket snabba, men DOM manipulation är i jämförelse väldigt långsamt och är ofta flaskhalsen i en webbapplikations prestanda (Zakas 2010).

Ett annat problem är att DOM baserar sig helt och hållet på mutabelt tillstånd. Detta gör det svårt att beskriva användargränssnitt deklarativt, och leder till en hel del kod som måste hålla reda på tillståndet.

2.2 Exempel: Användargränssnitt för en musikdatabas

Låt oss ta undersöka ett mycket enkelt exempel för att demonstrera problemet med mutabelt tillstånd. Vår påhittade applikation ska visa en tabell på populära musikalbum.

Först vill vi visa albumen i en tabell:

```
var albums = getAlbumsFromDatabase();

var $tableOfAlbums =
    $('<table>').append(
        $('<thead>').append(
            $('<tr>').append(
                $('<th>').text('Album name'),
                $('<th>').text('Artist'),
                $('<th>').text('Year'),
                $('<th>').text('Rating'),
            )
        ),
        $('<tbody>').append(
            $.map(albums, function(album) {
                return $('<tr>').append(
                    $('<td>').text(album.title),
                    $('<td>').text(album.artist),
                    $('<td>').text(album.year),
                    $('<td>').text(album.rating),
                    $('<td>').append(
                        $('<a>').attr('href', '#link').text('Read...')
                    )
                )
            })
        )
    );

$('body').append($tableOfAlbums);
```

Figur 1. Imperativ musikdatabas del 1

Koden i figur 1 bygger den tidigare nämnda första representationen av vyn. Till nästa vill vi göra det möjligt för användaren att ta bort ett album från listan:

```
function removeAlbum(event) { // Tillagd
    $(event.target).parents('tr').remove();
}

var $tableOfAlbums =
    $('<table>').append(
        $('<thead>').append(
            $('<tr>').append(
                $('<th>').text('Title'),
                $('<th>').text('Artist'),
                $('<th>').text('Year'),
                $('<th>').text('Rating'),
                $('<th>').text('Remove') // Tillagd
            )
        ),
        $('<tbody>').append(
            $.map(albums, function(album) {
                return $('<tr>').append(
                    $('<td>').text(album.title),
                    $('<td>').text(album.tagline),
                    $('<td>').text(album.year),
                    $('<td>').text(album.rating),
                    $('<td>').append(
                        $('<a>').attr('href', '#link').text('Read...')
                    ),
                    $('<td>').append(
                        // Tillagd
                        $('<button>').text('Remove').click(removeAlbum)
                    )
                )
            })
        )
    );
// ...
```

Figur 2. Imperativ musikdatabas del 2

Hittills är exemplet mycket enkelt, men också orealistiskt. På riktigt skulle data antagligen fås och manipuleras via förfrågan till en server. Till nästa vill vi simulera att `removeAlbum` funktionen gör en förfrågan till en server för att ta bort albumet från listan. Vi låter användargränssnittet göra en optimistisk uppdatering, dvs. albumet tas bort från listan före servern har svarat. Då förfrågan misslyckas måste listan uppdateras så att albumet läggs tillbaka till sin ursprungliga position.

```
// ...
function simulateAsyncDeleteRequest() { // Tillagd
    var def = $.Deferred();

    // förfrågan misslyckas efter 1 till 5 sekunder
    _.delay(def.reject, _.random(1000,5000));

    return def.promise();
}

function removeAlbum(event) { // Modifierad

    var $rowToRemove = $(event.target).parents('tr');
    var $rowBelowRemoved = $rowToRemove.next();

    // optimistisk UI uppdatering
    $rowToRemove.remove();

    // annulera uppdateringen då förfrågan misslyckas
    simulateAsyncDeleteRequest().fail(function() {
        $rowBelowRemoved.before($rowToRemove);
    });
}
// ...
```

Figur 3. Imperativ musikdatabas del 3

På första intrycket kan koden i figur 3 verka fungerande, men det finns flera problem med den. Då vi tar bort ett element från listan, sparar vi med samma en referens till elementet under den så att då uppdateringen misslyckas kan elementet lätt läggas tillbaka på dess gamla plats. Men vad händer då vi försöker ta bort sista elementet i listan?

Vi kunde lätt ta hand om detta specialfall, men det finns en mycket seriösare brist med exemplet: det kan finnas flera asynkrona förfrågan till servern i gång samtidigt. Vad händer ifall vi tar bort två på varandra följande element från listan? Då kommer den första inte att kunna placeras tillbaka i listan eftersom den berodde på elementet under sig.

Det tredje problemet är att då vi tar bort och lägger tillbaka ett element, kommer inte ”ta bort” knappen att fungera eftersom dess event listener också togs bort. Detta kunde åtgärdas genom att ta hand om knapparnas events högre upp i DOM hierarkin, men det är ändå en bugg som inte är alldeles självklar.

Poängen här är att då man håller på med sådan här DOM manipulering är det lätt att orsaka buggar som inte visar sig genast. I vårt exempel kunde vi försöka bygga någon

sorts register som håller reda på halvfärdiga förfrågan och logik som placerar elementen tillbaka på rätt plats. Men låt oss istället pröva något enklare.

Låt oss modifiera `removeAlbum` funktionen så att den inte tar bort element före servern har svarat. Istället kan vi först gömma elementet med CSS först och ta bort den helt och hållet då servern svarar.

```
function removeAlbum(event) { // Modifierad

    var $rowToRemove = $(event.target).parents('tr');

    // optimistisk UI uppdatering
    $rowToRemove.hide();

    // visa elementet igen ifall förfrågan misslyckas
    simulateAsyncDeleteRequest().fail(function() {
        $rowToRemove.show();
    }).done(function() {
        $rowToRemove.remove();
    });
}
```

Figur 4. Imperativ musikdatabas del 4

Nu borde annuleringsmekansismen fungera rätt även ifall man tar bort flera element i följd. Men vad om vi använde CSS för att skapa ränder i listan så att radernas färg alternerade mellan två olika färger? Isåfall skulle ränderna gå sönder medan vi väntar på svar från servern och elementet mellan två rader med samma färg göms.

Vi kunde fixa detta genom att lägga en temporär rad bredvid den gömda raden, men det vidare öka mängden kod och skulle fortsätta göra vår vy-logik mer komplex. Det uppstår ett klart mönster av växande komplexitet som följd av manuell DOM manipulation.

Låt oss försöka lägga till en sista funktionalitet i vårt exempel: en numerisk indexering för varje rad.

```
// ...
var $tableOfAlbums =
    $('<table>').append(
        $('<thead>').append(
            $('<tr>').append(
                $('<th>').text('Index'), // Tillagd
                $('<th>').text('Title'),
                $('<th>').text('Artist'),
                $('<th>').text('Year'),
                $('<th>').text('Rating'),
                $('<th>').text('Remove')
            )
        ),
        $('<tbody>').append(
            $.map(albums, function(album, index) { // Modifierad
                return $('<tr>').append(
                    $('<td>').text(index), // Tillagd
                    $('<td>').text(album.title),
                    $('<td>').text(album.tagline),
                    $('<td>').text(album.year),
                    $('<td>').text(album.rating),
                    $('<td>').append(
                        $('<a>').attr('href', '#link').text('Read...')
                    ),
                    $('<td>').append(
                        $('<button>').text('Remove').click(removeAlbum)
                    )
                )
            })
        )
    );
// ...
```

Figur 5. Imperativ musikdatabas del 5

Ifall vi nu tar bort en rad från tabellen kommer numreringen att bli osynkroniserad. Att lägga till logik för att ta hand om detta skulle åter leda till mer komplexitet i vårt exempel.

Finns det någon lösning till problemen vi hittills stött på? Det verkar som om vårt exempel fungerar riktigt fint för att skapa den första representationen av tabellen, men leder till problem genast då vi börjar modifiera DOM strukturen.

Vad om vi istället för inkrementella uppdateringar istället byggde hela tabellen på nytt varje gång något ändras? Då skulle tabellen vara i ett perfekt tillstånd som första representationen hittills varit, hela tiden.

```
var albums = getAlbumsFromDatabase();

function simulateAsyncDeleteRequest() {
    var def = $.Deferred();
    _.delay(def.reject, _.random(1000,5000));

    return def.promise();
}

function removeAlbum(rowId) {
    var albumToRemove = _.findWhere(albums, { id: rowId });

    albums = _.reject(albums, function(album) { return album.id === rowId; });
    renderAlbumTable(albums);

    simulateAsyncDeleteRequest().fail(function() {
        albums = albums.concat(AlbumToRemove);
        renderAlbumTable(albums);
    });
}

function renderAlbumTable(albumData) {
    var albumsSortedById = _.sortBy(albumData, 'id');
    $('body').empty().html(
        $('<table>').append(
            $('<thead>').append(
                $('<tr>').append(
                    $('<th>').text('Title'),
                    $('<th>').text('Artist'),
                    $('<th>').text('Year'),
                    $('<th>').text('Rating'),
                    $('<th>').text('Remove')
                )
            ),
            $('<tbody>').append(
                $.map(albumsSortedById, function(album, index) {
                    return $('<tr>').append(
                        $('<td>').text(album.title),
                        $('<td>').text(album.tagline),
                        $('<td>').text(album.year),
                        $('<td>').text(album.rating),
                        $('<td>').append(
                            $('<a>').attr('href', '#link').text('Read...')
                        ),
                        $('<td>').append(
                            $('<button>')
                            .text('Remove')
                                .click(_.partial(removeAlbum, album.id))
                        )
                    )
                })
            )
        )
    );
}
renderAlbumTable(albums);
```

Figur 6. Imperativ musikdatabas del 6

I koden i figur 6 gör vi inte längre någon manuell DOM manipulering, utan vi bygger en helt ny DOM struktur vid varje förändring. Denna strategi löser alla problem vi stött på hittills. Vi behöver inte längre skriva kod vars enda uppgift är att hålla datat och vyn synkroniserade, eftersom vyn alltid byggs som en funktion av det uppdaterade datat.

Det finns dock ett problem med denna strategi: Som det nämndes i början av kapitlet är DOM manipulering mycket långsamt i jämförelse med tolkning av JavaScript. Att bygga hela vyn på nytt vid varje förändring blir snabbt för långsamt då man går vidare från små exempel till riktiga applikationer.

Men vad om det var möjligt att automatiskt göra den minsta mängden DOM manipulation som krävs för att bygga om vyn? Detta är exakt var JavaScript biblioteket React gör.

3 REACT

React är ett JavaScript bibliotek med öppen källkod som utvecklats av Facebook. React började som ett internt projekt hos Facebook och publicerades till allmänheten första gången år 2013. Sen dess har React blivit ett av de populäraste JavaScript biblioteken och används av företag som Facebook, Instagram, Netflix, Imgur och Airbnb (React.js 2016).

React är ett deklarativt vy-bibliotek som låter programmeraren definiera hur applikationen ska se ut under en punkt i tiden, medan React sköter om att användargränssnittet automatiskt uppdateras då det underliggande datat förändras. Till skillnad från ramverk sköter React endast vy-delen av applikationen, och tar inte ställning till hur resten av applikationen struktureras. Det har dock dykt upp flera populära sätt att strukturera React applikationer varav många bygger på arkitekturmönstret Flux som används och utvecklades av Facebook (Flux 2016).

De grundläggande idéerna bakom React är Virtual DOM och komponenter. Låt oss gå noggrannare in på hur dessa används i biblioteket.

3.1 Virtual DOM

I förra kapitlet granskades imperativ DOM manipulering och dess problem. Slutsatsen var att många problem kunde undvikas ifall vi byggde om hela vyn varje gång datamodellen uppdaterades. Problemet med denna strategi var dock att även om JavaScript motorer nuförtiden är mycket snabba är DOM manipulering i kontrast mycket långsamt.

React löser detta problem m.h.a virtuell DOM (VDOM). VDOM är en representation av DOM hierarkin i minnet. Tack vare JavaScript motorernas snabbhet kan denna representation manipuleras mycket effektivt. Efter att VDOM har uppdaterats i respons till nytt data, körs sedan en algoritm som räknar ut skillnaden mellan den nya representationen och riktiga DOM, och bestämmer den minsta mängden DOM manipulation som krävs för att synkronisera DOM med den virtuella representationen (React.js 2016).

Tack vare VDOM kan programmeraren beskriva en vy deklarativt, som om den byggdes från början vid varje uppdatering, och låta diffing algoritmen sköta om att förändringarna görs effektivt. Programmeraren behöver inte längre hålla reda på att en komponents abstrakta representation och dess manifestation i DOM hålls synkroniserade. Detta tar bort behovet för fel benägen tillståndsmanipulering som programmeraren annars måste sköta om.

3.2 Komponenter

React applikationer struktureras som en hierarki av React-komponenter. En React-komponent är en självstående och återanvändbar komponent som innehåller all logik om hur den ska se ut med given inmatning.

Den viktigaste delen av en komponent är dess `render` metod. `render` metoden är en ren funktion från `state` och `props` till VDOM, dvs. med samma `state` och `props` returnerar funktionen alltid samma VDOM. En komponents `render` metod körs automatiskt varje gång komponentens `props` eller `state` ändras.

```
var React = require('react');

module.exports = React.createClass({
    _incrementLikes: () => {
        this.setState({
            likes: this.state.likes + 1
        });
    },
    getInitialState: () => {
        return {
            likes: 0
        };
    },
    render: () => {
        return (
            <div className="like-counter-component">
                <input type="button"
                    onClick={this._incrementLikes}
                    value={this.state.likes} />
            </div>
        );
    }
});
```

Figur 7. En "Like knapp" React komponent

I exemplet ovan returnerar komponentens `render` metod JSX, en XML liknande syntax som transformeras till kapslade funktionsanrop till React biblioteket. JSX snutten i exemplet ovan transformeras till följande funktionsanrop:

```
React.createElement(
    "div",
    { className: "like-counter-component" },
    React.createElement("input", {
        type: "button",
        onClick: this._incrementLikes,
        value: this.state.likes
    })
);
```

Figur 8. Omvandlad JSX

JSX är valfritt att använda men är populärt eftersom det hjälper visualisera hierarkin bland elementen.

3.3 Lifecycle metoder

Förutom den obligatoriska `render` metoden har en React komponent ett antal speciella metoder som körs under specifika tider under komponentens livslängd (React.js 2016):

- `getInitialState` körs en gång före komponenten monteras i DOM och returnerar det första `state` objektet.
- `componentWillMount` körs en gång före första körningen av `render` metoden.
- `componentDidMount` körs då komponenten har monterats i DOM. I denna metod har man tillgång till komponentens DOM element.
- `componentWillReceiveProps` körs varje gång komponenten mottar ett nytt `props` objekt från sin förälder komponent. Funktionen får som parameter det nya `props` objektet.
- `componentWillUpdate` körs varje gång komponenten mottar ett nytt `state`- eller `props` objekt. Funktionen får som parameter de nya objekten.
- `componentDidUpdate` körs genast efter uppdateringar har spolats till DOM.
- `componentWillUnmount` körs strax före komponenten tas bort från DOM.

M.h.a lifecycle metoderna kan programmeraren få fast viktiga skeden i komponentens livslängd och se till att viss kod körs rätt tid. T.ex. så måste man i `componentWillUnmount` metoden se till att man ”städar undan” eventuella DOM element, event listeners eller annan initialisering som gjorts i `componentDidMount`.

Trots att lifecycle metoder ger programmeraren noggrann kontroll över komponentens livslängd så består speciellt mindre komponenter ofta endast av en `render` metod. Komponenter som endast består av en `render` metod kallas ”rena” komponenter eftersom de är rena funktioner från data till VDOM. Det är nyttigt att försöka hålla sig till rena komponenter så mycket som möjligt eftersom de är enkla att förstå, debugga och testa.

4 FUNKTIONELL REAKTIV PROGRAMMERING

FRP är en programmeringsparadigm som beskriver asynkrona händelser och värden som ändar med tiden deklarativt och på hög nivå (Wan, Hudak 2000). FRP bygger på `EventStreams` som representerar en sekvens av värden över tid. `EventStreams` kan manipuleras med funktioner som `map` och `filter`, och kombineras med kombinatorer för att skapa nya `EventStreams`. `EventStreams` är observerbara, dvs. det är möjligt att lyssna på en `EventStream` och reagera till nya värden.

På samma sätt som funktionell programmering lyfter nivån av abstraktion från imperativ iteration över en samling av värden till högre nivås funktioner som `map`, `filter` och `reduce` så gör FRP samma för `Events`. Istället för att behandla enskilda `Events` jobbar man i FRP på en högre abstraktionsnivå m.h.a `EventStreams`.

4.1 Bacon.js

Bacon.js är ett populärt FRP bibliotek för JavaScript vars starka sidor är liten storlek och fokus på korrekt kör ordning av beräkningar (Bacon.js 2016). Bacon.js är utvecklad av min kollega Juha Paananen från Reaktor.

Bacon.js hämtar med sig två typer av `Observable`: `EventStream` och `Property`.

4.1.1 EventStream

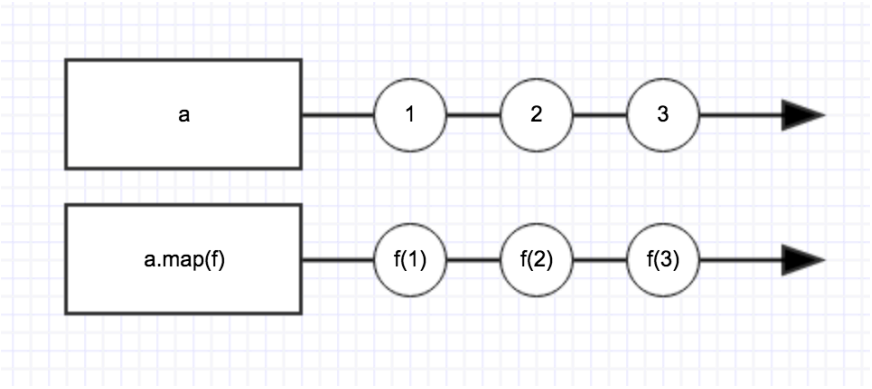
`EventStreams` nämndes i början av detta kapitel, och Bacon.js `EventStreams` uppfyller denna specifikation. Det är alltså frågan om en representation av en sekvens av diskreta värden över tid, som man kan koppla en eller flera `Subscribers` till. En `Subscriber`-funktion körs för varje nytt värde i strömmen. Ifall en `EventStream` inte har en enda lyssnare försvinner dess värden i cyberrymden.

En Property är exakt som en `EventStream` förutom en viktig skillnad: den har ett nuvarande värde. Medan en `EventStream` består av en sekvens diskreta värden över tid beskriver en `Property` ett värde som ändrar med tiden. Detta betyder att ifall man läg-

ger en lyssnare på en `Property` får lyssnaren genast det nuvarande värdet, istället för att först få nästa värde.

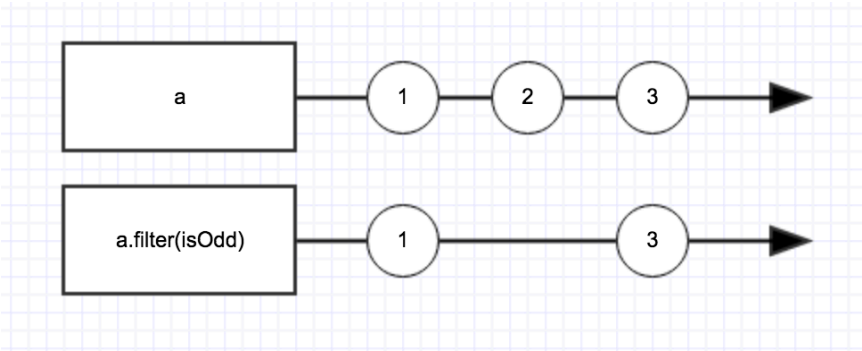
4.1.2 Viktiga metoder

Under finns en kort beskrivning på de viktigaste metoderna i Bacon.js.



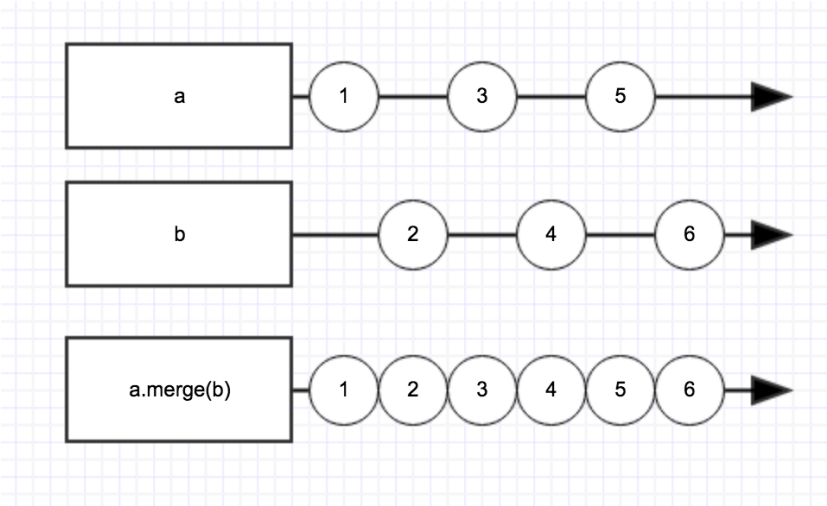
Figur 9. Bacon map

`map` – returnerar en ny `Observable` som är resultatet av att applicera en funktion `f` på varje värde i den ursprungliga `Observable`.



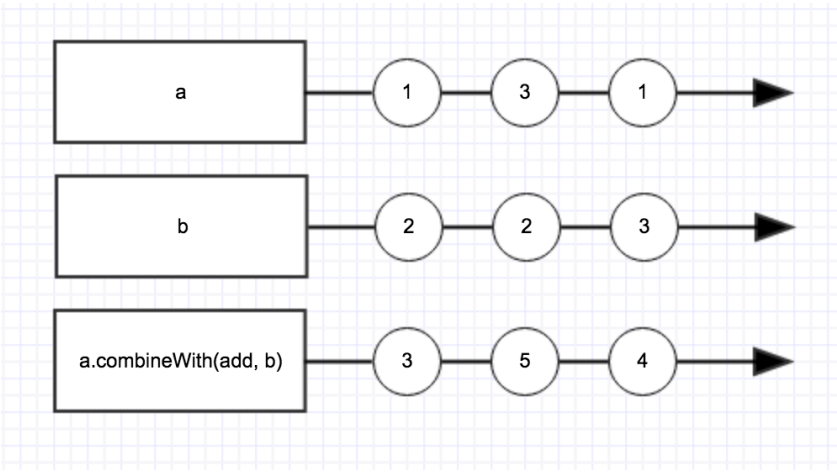
Figur 10. Bacon filter

`filter` – returnerar en ny `Observable` som består av alla värden i den ursprungliga `Observable` som predikatfunktionen `f` returnerar sanna värden för.



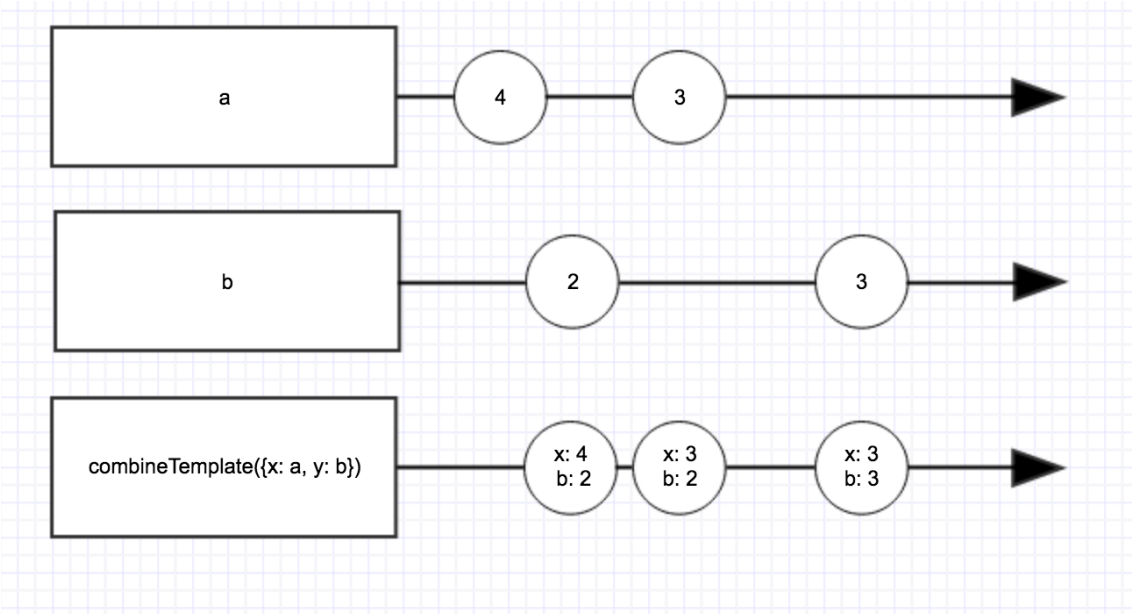
Figur 11. Bacon merge

`merge` – en kombinator som returnerar en `EventStream` som innehåller värden från både den ursprungliga `Observable` och den som getts som parameter åt `merge`.



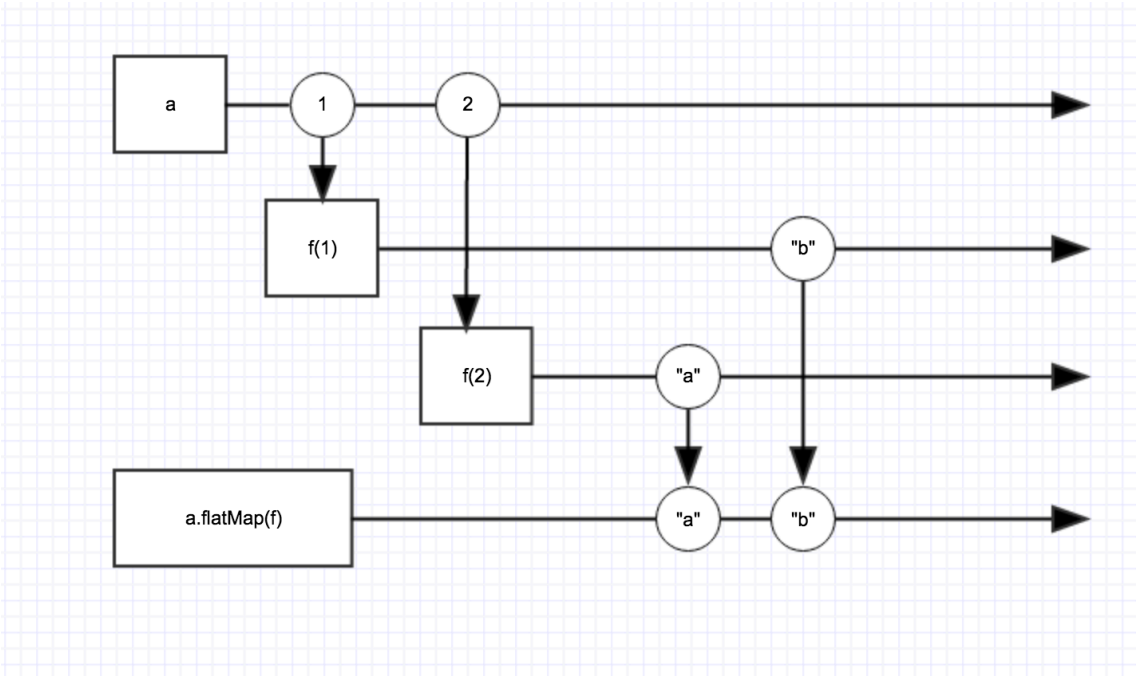
Figur 12. Bacon combineWith

`combineWith` – en kombinator som returnerar en ny `Observable` som består av resultatet av att applicera en funktion `f` över senaste värde i två eller flera `Observable`.



Figur 13. Bacon combineTemplate

`combineTemplate` – är en kombinator som tar ett JavaScript objekt bestående av två eller flera nyckel-värde par, där värden är `Observable`. Returnerar en `Property` vars värden är JavaScript objekt som innehåller det nyaste värdet för varje `Observable` som gavs åt metoden.



Figur 14. Bacon flatMap

`flatMap` – applicerar en funktion `f` som returnerar en ny `Observable` - för varje värde i den ursprungliga `Observable`. Returnerar en `Observable` som innehåller alla värden i de enskilda `Observable` som `f` skapat.

4.2 Exempel: BMI kalkylator

I detta exempel används JavaScript biblioteket Bacon.js för att implementera en BMI-kalkylator användes FRP principer.

```
const $ = require('jquery');
const Bacon = require('baconjs');

function calculateBmi(m, kg) {
    return kg / Math.pow(m, 2);
}

const heightStream = $('#height-slider').asEventStream('change').map('.value');
const weightStream = $('#weight-slider').asEventStream('change').map('.value');

const bmiStream = combineWith(calculateBmi, heightStream, weightStream);

bmiStream
    .map((bmi) => {
        if (bmi < 18.5) { return 'underweight'; }
        else if (bmi < 25) { return 'normal'; }
        else if (bmi < 30) { return 'overweight'; }
        else { return 'obese'; }
    })
    .onValue((result) => {
        $('#result').text('You are: ' + result);
    });
```

Figur 15. Funktionell reaktiv BMI-kalkylator

Bacon.js lägger till `asEventStream` metoden på `jQuery` objektet. Metoden returnerar en `EventStream` som får ett nytt värde för varje `Event` av den specificerade sorten på ifrågavarande DOM element. Sedan används `map` för att få själva siffervärdet från `Event` objektet.

Kombinatorfunktionen `combineWith` tar som parameter en funktion `f` och n antal `EventStreams`, och returnerar en ny `EventStream` vars värden är resultatet av att applicera `f` på värdena från strömmarna. Genom att kombinera `heightStream` och `weightStream` med BMI-funktionen får vi `bmiStream` som innehåller BMI värdet för

nuvarande värdena av längd och vikt sliders. Till slut appliceras en funktion från BMI värde till viktkategori för varje nya BMI värde och resultatet skrivs i DOM.

Koden i exemplet har ett några egenskaper värda att nämna:

- Den beskriver ett asynkront system på ett deklarativt sätt
- Det sker ingen explicit tillståndsmanipulering
- Data flöder i en riktning
- Koden är på hög abstraktionsnivå vilket leder till koncishet

Dessa egenskaper gör FRP kod lätt att resonera om även i större och mer avancerade program.

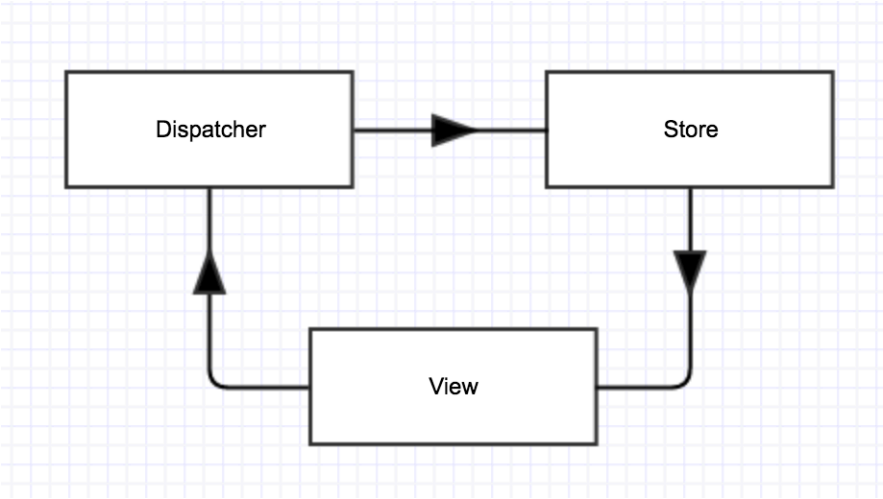
5 FLUX-INSPIRERAD FRONT-END ARKITEKTUR

Eftersom React endast sköter om vy-delen av en applikation är det upp till användaren att bestämma hur hon vill sköta resten av logiken. Då Facebook började använda React internt kom de på ett arkitekturmönster som de började kalla Flux (Flux 2016). Flux baserar sig på enkelriktat dataflöde och dess huvudmål är att skapa enkla, förståbara och testbara användargränssnitt.

Då Facebook publicerade Flux ledde det till flera alternativa implementationer, varav flera kombinerar React med ett FRP bibliotek som Bacon, Kefir eller Rx.js. Jag kommer att använda Bacon i kommande exempel.

5.1 Översikt

Arkitekturmönstret består på hög nivå av tre delar: `View`, `Dispatcher` och `Store`.



Figur 16. Flux arkitekturdiagram

5.1.1 View

`View` delen i vår arkitektur består av React komponenter vars uppgift är att bygga om användargränssnittet i respons till ändringar i applikationsdata och att meddela användarens aktioner till `Dispatcher` komponenten.

5.1.2 Dispatcher

`Dispatcher` komponenten tar användarens aktioner från `View` komponenten och exponerar `EventStreams` bestående av dem.

```
const Bacon = require('baconjs');

const actionBus = new Bacon.Bus();

function dispatch(action) {
    actionBus.push(action);
}

function actionStreamByAction(actionName) {
    return actionBus.filter((action) => action.name === actionName);
}

module.exports = {
    dispatch,
    actionStreamByAction,
}
```

Figur 17. Dispatcher komponent

Alla användarens aktioner går via `dispatch` metoden. Utanför `Dispatcher` komponenten kallar man sedan på `actionStreamByAction` metoden för att få en `EventStream` bestående av specifika aktioner.

5.1.3 Store

I `Store` modulen komponeras användarens aktioner och asynkron data från olika källor till applikationens tillstånd. Denna modul innehåller största delen av applikationslogiken i form av `Observables` som kombineras med och härleds från varandra med olika kalkylationer. Utåt exponeras `Observables` som innehåller det senaste tillståndet för en viss bit data. React komponenterna kan sedan lyssna på dessa och automatiskt bygga om vyn då datat ändras.

5.2 Exempel: Musikdatabas användes FRP

I första kapitlet diskuterades problemen med imperativ tillståndsmanipulering i DOM med hjälp av en exempelapplikation. Låt oss nu bygga om exemplet användes de teknologier och principer som tagits upp hittills.

```
import _ from 'lodash';
import React from 'react';
import Bacon from 'baconjs';
import albumStore from 'album-store';
import dispatcher from 'dispatcher';

module.exports = React.createClass({
    _removeAlbum: function(id) {
        dispatcher.dispatch('removeAlbum', id);
    },

    componentDidMount: function() {
        albumStore.albumsDataProperty
            .takeWhile(this.isMounted.bind(this))
            .onValue(data => {
                this.setState({ albums: data });
            });
    },

    render: function() {
        const that = this;

        function renderAlbumsList(albums) {
            return _.map(albums, (album, index) => {
                return (
                    <div className="album-container">
                        <div>{index}</div>
                        <div>{album.title}</div>
                        <div>{album.artist}</div>
                        <div>{album.year}</div>
                        <div>{album.rating}</div>
                        <input type="button" value="Remove album"
                            onClick={_.partial(that._removeAlbum, album.id)} />
                    </div>
                );
            });
        }
        function renderSpinner() {
            return (
                <div className="spinner"></div>
            );
        }

        return (
            <div className="page-container">
            {
                this.state.albums
                    ? renderAlbumsList(this.state.albums)
                    : renderSpinner()
            }
            </div>
        );
    }
});
```

Figur 18. View komponent

Figur 18 är React klass som fungerar som `View` komponenten i vårt exempel. Komponenten lyssnar på `albumsDataProperty` som alltid innehåller det senaste informationen om albumen som ska visas. Märk `takeWhile` metoden på `albumData-`

`Property` som ser till att lyssnaren inte blir kvar då komponenten avmonteras från DOM.

I komponentens `render` metod finns två hjälpfunktioner varav `renderAlbumsList` returnerar VDOM strukturen för albumtabellen och `renderSpinner` returnerar VDOM för en laddningsindikator. I slutet av `render` metoden kallas endera av dessa hjälpfunktioner beroende på om albumdata existerar. Varje gång `albumsData-Property` får ett nytt värde uppdateras komponentens `state` objekt som leder till att `render` metoden körs om och vyn uppdateras med nyaste data.

Varje album i listan har en knapp för att ta bort albumet från listan. Då en sådan knapp trycks kallas `Dispatcher` komponentens `dispatch` metod med albumets id som parameter.

Det finns några saker värda att notera i figur 18. En viktig poäng är att koden är mycket enkel och deklarativ. Det finns ingen tillståndshanteringskod i hela komponenten, utan koden endast beskriver hur vyn ska se ut med given applikationsdata. Eftersom hela komponenten i essens omvandlar applikationsdata till VDOM är koden lätt att förstå och resonera om, och har ett klart avgränsat ansvarsområde. Detta minskar programmerarens kognitiva belastning och låter henne koncentrera sig på själva problemet. En annan viktig poäng är att data flöder i en riktning: data flöder in från `albumStore` och ut i form av användarens aktioner via `Dispatcher`.

```
import Bacon from 'baconjs';
import dispatcher from 'dispatcher';

const albumRemovedStream = dispatcher.actionStreamByAction('removeAlbum')
    .map('.data')
    .flatMap(albumId => {
        return Bacon.fromPromise(
            fetch('/api/albums/remove/' + albumId, { method: 'POST' })
        );
    })
    .filter(response => response.status === 200);

const albumDataProperty = Bacon.once(true)
    .merge(albumRemovedStream)
    .flatMap(val => {
        return Bacon.fromPromise(
            fetch('/api/albums').then(response => response.json())
        );
    })
    .toProperty();

module.exports = {
    albumDataProperty
};
```

Figur 19. Store komponent

I figur 19 ser vi Store-komponenten som innehåller den huvudsakliga applikationslogiken. `albumRemovedStream` lyssnar på `removeAlbum` aktioner via `Dispatcher`-komponenten och gör en förfrågan till servern för att ta bort ett album. `albumRemovedStream` får ett värde ifall förfrågan lyckas med http kod 200.

`albumDataProperty` ber album-data från servern en gång då sidan laddas och varje gång ett album tagits bort. Detta leder till att `albumDataProperty` alltid innehåller senaste album-data. Notera att koden än en gång är funktionell och deklarativ utan någon imperativ tillståndshantering.

5.3 Sammanfattning

Denna applikationsarkitektur är inte endast enkel utan också mycket komponerbar. Användargränssnittet kan byggas med isolerade och återanvändbara React-komponenter och applikationslogiken kan delas in i flera `Store` komponenter varav varje har ett klart domänbundet ansvarsområde. Varje View komponent lyssnar på ändringar i data som angår den. Data flöder alltid i en riktning vilket gör att koden är lätt att förstå och testa.

Dessutom är detta mönster oberoende av de enskilda biblioteken: varje bibliotek kan bytas ut mot en motsvarande utan större ändringar i applikationslogiken.

6 SLUTSATSER

Webben har blivit en applikationsplattform och då webbapplikationer blivit större och mer komplicerade har teknologierna varit tvungna att anpassa sig. Imperativ vy-manipulering är ett dåligt verktyg för att bygga användargränssnitt i avancerade applikationer med mycket asynkrona händelser. React och VDOM har möjliggjort reaktiv programmering på webben vilket är ett stort framsteg som underlättar skapandet av avancerade användargränssnitt på denna plattform. Arkitekturmönstret som presenterades i senaste kapitlet har möjliggjorts av dessa framsteg.

Målet med arbetet var att presentera detta arkitekturmönster samt ge läsaren en förståelse om vad som gör FRP ett effektivt sätt att skapa användargränssnitt. Mina personliga erfarenheter med denna arkitektur har visat att en applikation byggd med dessa principer kan öka i storlek utan att öka i komplexitet.

7 KÄLLOR

Document Object Model. 2016. Tillgänglig:
https://en.wikipedia.org/wiki/Document_Object_Model
Hämtad 27.4.2016.

jQuery. 2016. Tillgänglig: http://jquery.com/
Hämtad 27.4.2016.

Zakas, Nicholas C. 2010, High Performance JavaScript. Tillgänglig:
http://cdn.tsq.me/ebook/High%20Performance%20JavaScript.pdf
Hämtad 27.4.2016.

React.js. 2016. Tillgänglig https://facebook.github.io/react/
Hämtad 27.4.2016.

Flux. 2016. Tillgänglig https://facebook.github.io/flux/
Hämtad 27.4.2016.

Wan, Hudak. 2000. Functional Reactive Programming from First Principles.
Tillgänglig: http://haskell.cs.yale.edu/wp-content/uploads/2011/02/frp-1st.pdf
Hämtad 27.4.2016.

Bacon.js. 2016. Tillgänglig: https://github.com/baconjs/bacon.js
Hämtad 27.4.2016.