Inläsning

- Algoritmgruppen
  - Infrastruktur
  - Data set
  - Scheduling
- Front end
  - Jasmine (TG) (lite i labbar)
  - Bootstrap
  - DOM
  - JavaScript stilguide (SG?)
- Back end
  - Jersey client (TG)
  - Jersey server
  - H2

Grundsystemet
ETSN05

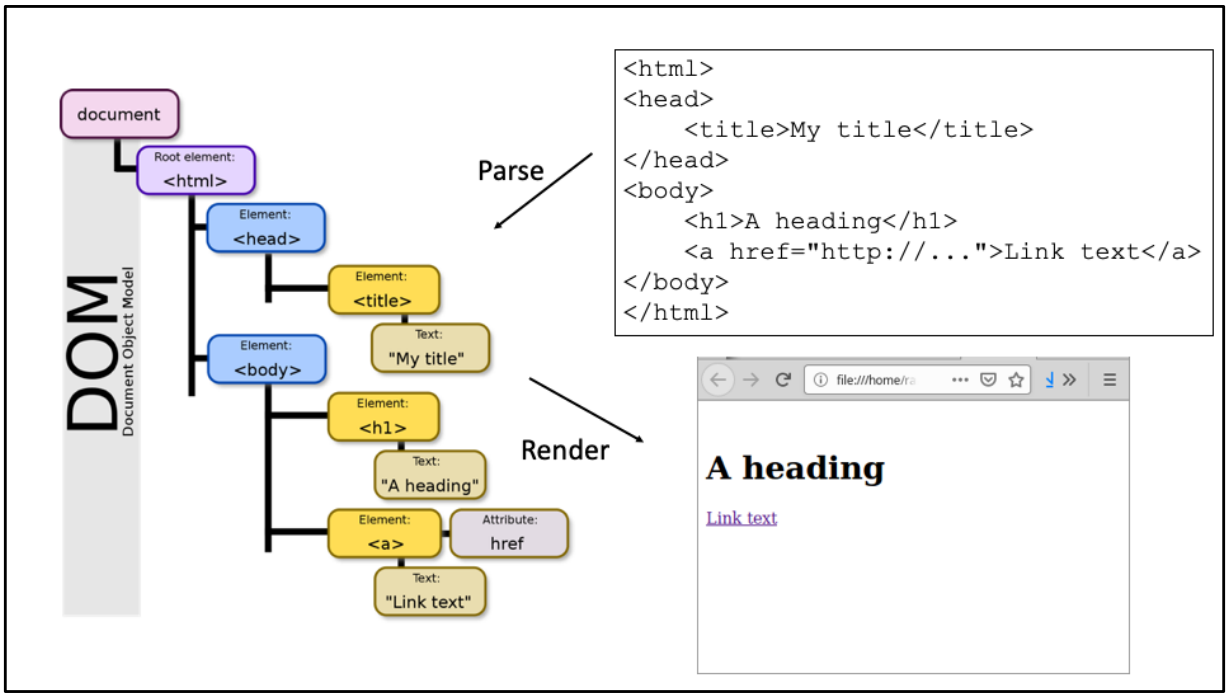
JavaScript, DOM

Rasmus.ros@cs.lth.se

Web front-end programming

- Interface defined in HTML
- Style and layout modified in CSS
- Dynamic behaviour in JavaScript

Tre komponenter idag
Framförallt JavaScript



HTML beskriver DOM trädet
HTML parsas av webläsaren till DOM träd
DOM trädet renderas av webbläsaren
Ser inget JS eller CSS här

Document object model (DOM)

- Is the actual web interface
- JavaScript API for modifying

- Nodes – one per HTML tag
- Handle events – keyboard/mouse, loading state

DOM objekt trädet är gränssnittet
Man använder JS för att ändra I DOM trädet
Mål: portabel JS

Olika typerna av DOM objekt som kommer användas
Events, sätt att lyssna på förändringar

Website stiles

- Classic website
  - Everything in HTML
  - Wikipedia, news sites?, blogs?
- Hybrid website
  - Server returns HTML with JavaScript updates
  - Twitter, YouTube
- Single page application (SPA)
  - Server returns data and renders with JavaScript
  - One page load
  - Gmail, Office365?, BASE

Olika stilar på hemsidor
Hybrid vanligt, bästa och sämsta av båda världar
BASE är en SPA
Tydlig ägarskap
Lite klumpigt att göra vissa saker

HTML

- Text content

```
<h1>Title</h1>
<p>Paragraph</p>
```

- Basic layout

```
<div></div>
<span></span>
```

- Input

```
<form>
    <input type="text">
    <button>Submit</button>
</form>
```

Hinner bara täcka en bråkdel, tillräckligt för det som används under föreläsningen

Vanligt text innehåll bör läggas inom h1 och p

Div och span används för att gruppera innehåll.
Div gör radbrytning men span gör det inte

Input

- Lägg det under form för att hanteras tillsammans
- Reset, submit som en enhet
- Finns även drop down, checkbox, och radio button som är lätta att använda
- Gör inget utan att man använder javascript (eller onsubmit på form)

Cascading style sheets (CSS)

- Selectors specify what to affect
  - `button` – Tag name selector
  - `.btn-primary` – Class attribute selector
  - `#my-id` – Id attribute selector
- Combination: `button.btn.btn-primary`
- Hierarchy: `form button.btn`

- Use Bootstrap in project

```
<button class="btn btn-primary">
```

Återigen hinner vi bara gå igenom en bråkdel
Selektorer används för att hämta ut element från DOM trädet med CSS och JavaScript

Bootstrap är ett CSS bibliotek som stilar komponenterna
Man specifierar klass attribut som är definierat I Bootstrap
Ganska stort men man kan använda det man behöver genom deras dokumentation

CSS example

```
<style>
    .a { color: green; }
    .a .b {color: red; }
</style>
<div class="a">
    <p>green text</p>
    <p class="b">red text</p>
</div>
```

Kommer inte behöva lära er CSS förutom selektorerna
Prioritet påverkar stilen som läggs på
Grön färg på alla element som har typen a
Både a och b blir röd text, mer specifikt

Ni som har använt JavaScript innan, försök att inte gå för långt utanför det ni ser idag

JavaScript (ECMAScript)

```
const a = 1;
a = 2; // error

let b = 2.1;
b = "string";

const add = function(n1, n2) {
    return n1 + n2;
};

add(4, 1);
 > 5

add("a", 1); // Types?? Who cares?
 > a1
```

ECMAScript heter standarden
Använd version 6 av standarden

Dynamiskt programmeringsspråk

C-familjen: {} måsvingar och semikolon

Variabler kan vara funktioner (som add)

3 sätt att deklarera typer: const/let/var
Const och let används för det mesta

- var när man behöver deklarera saker flera gånger. (sällsynt). Konstig scope, undvik.
- let när referensen behöver skrivas över
- const annars

JavaScript in action

test.html

```
<script>
function show() {
    const myText = document.getElementById("myText");
    let userInput = myText.value;
    userInput = userInput.toUpperCase();
    alert(userInput);
  }
</script>
<input type="text" id="myText">
<button onclick="show()">Click</button>
```

Document är en global variabel som är ingången till DOM
getElementById hämtar ut en HTML tag som har blivit annoterad med id="myText"
Input är en DOM Node som har en value

JavaScript objects

```
const map = {
    key1: 1,
    'key 2': "str"
};
Map.key1
> 1
map['key 2']
> string
```

Ungefär som Map I Java (finns Map I Javascript också men går inte igenom det)
Använd sätt 1 för explicit namngivna variabler
Använd sätt 2 för godtycklig data

Kan nästlas

Functions

```
const standard = function(x, y) {}
const arrow = (x, y) => {};

const obj = {f: standard};
obj.f("x", 2);

standard("x", 2);
standard("x");  // y = undefined
standard();     // x and y = undefined
```

Finns några olika sätt att definiera funktioner
Vad som kallas standard är så vi rekommenderar här
Arrow funktionen är vanligt när man vill ha en anonym funktion

Kan använda variabler som pekar på funktioner
Spara dom I objekt och anropa dom som metoder I Java
Det går inte att ha flera funktioner med samma namn I samma scope
Det spelar ingen roll vilka parametrar en funktion anropas med, anropet kommer alltid till samma funktion
För att implementera valfria parametrar kan man kolla parametern efter undefined med typeof operatorn

JavaScript classes

```
const Car = function(name) {
    this.name = name;
    this.trademark = function() {
        return this.name + "™";
    };
};


const c = new Car('Tesla Model X');
c.trademark()
> Tesla Model X™
c.name
> Tesla Model X
```

Finns många sätt att definiera klasser
Här är sättet som jag rekommenderar
Car kallas en constructor function
this blir tillgängligt om man anropar funktionen med new, annars är this undefined (I strict mode)
Använder inte arv eller interfaces

Modules

```
var base = base || {};
base.foo = (function() {
    const priv = function() {};
    const public = {};
    public.x = y;
    return public;
})();
```

```
var base = base || {};
```

```
var base;
if (!base)
   base = {};

// IIFE
(function() { ... })();
```

```
function runOnce() { ... }
runOnce();
```

Namespace (base)
Det som vill exponeras är explicit
Man kan definiera samma namespace
Var används här för att man kanske behöver definiera base I flera filer och man vet inte vilken som laddas först
Let tillåter bara en definition

Control structures

```
for (let i = 0; i < 4; i++) {
    console.log(i);
}


let a = "echo";
while (a) {
    console.log(a);
    a = a.slice(1);
}


if (x) {}
else {}
```

Det mesta som I Java

While loopen kommer exekveras tills strängen är tom, konceptet kallas truthy/falsy
0, false, tom sträng, undefined är falsy, allt annat är truthy

Arrays

```
let arr = ["a", "b", "c"];


arr.slice(1, 3);
> ["b", "c"]


arr.push("d");
const d = arr.pop();
const a = arr.shift();
arr.unshift(a);
```

Array looping

```
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
arr.forEach(console.log); // almost same


arr.map(function(x) {
    return {letter: x};
});
> [{letter: "a"}, {letter: "b"}, {letter: "c"}, {letter: "d"}]
```

Man kan göra operationer på arrays smidigt genom att skicka in en funktion som parameter
forEach gör något med varje element, tex. console.log loggar vad den anropas med
Map applicerar funktionen som den tar in och skapar en ny lista

Concurrency

```
setTimeout(function() {
    console.log("second");
}, 0);
console.log("first");
> first
> second

const data1 = myAsyncMethod(); // Not possible
const data2 = mySlowSyncMethod(); // Not done
```

JavaScript I webbläsaren har bara en tråd
setTimeout hör till DOM APIet
Även om tiden är satt till 0 så är det garanterat ordningen blir rätt
En funktion kör alltid helt färdigt innan asynkrona anrop startar
Så är det inte I Java

Funktionen som skickas in till setTimeout kallas callback
När asynkrona funktioner ska "returnera" data så skickas det med till en parameter I funktionen

Promise and fetch

```
fetch('/rest/foo')
    .then(function(response) {
        if (response.ok) return response.json();
        else throw Error('Got ' + response.status);
    }).then(function(foos) {
        // Do something with foos
    }).catch(function(error) {
        alert(error);
    });
base.rest.getFoos(function(foos) { /* ... */ }
```

Fetch används för att göra HTTP anrop asynkront
Fetch metoden returnerar objekt av typen promise
Promise är en vidareutveckling på callbacks
Promise kan kedjas, response.json returnerar ett nytt promise
Använd then och catch
Funktionerna som skickas med till then och catch får response objekt som parameter

Rest.js definierar alla rest anrop som är tillgängliga I BASE

DOM access

- **Root node:** `document`
- **Node**
  - `children, parent`
  - `querySelector('sel')`
  - `querySelectorAll('sel')`
  - `getElementById('myId')`

DOM dynamic content

- InnerHTML

```
const url = '/page';
document.getElementById('my-div').innerHTML =
    '<a href=' + url + '>Link</a>';
```

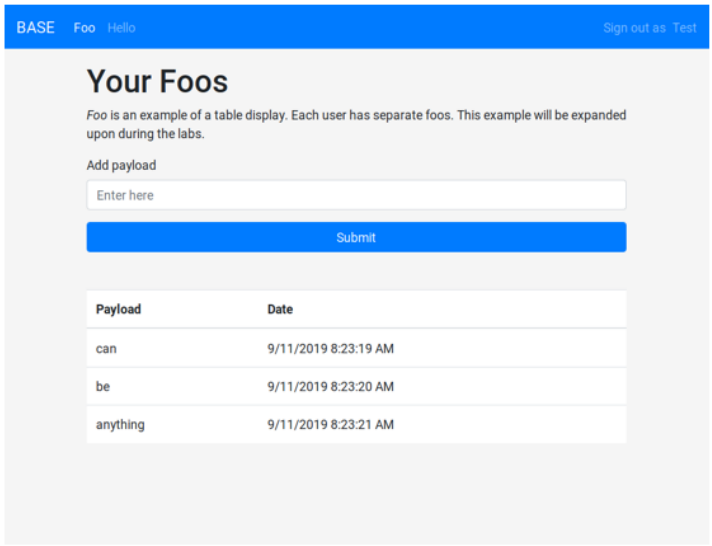
- Template
  - Complex
  - Structured, not "string programming"
  - See next section
- Event handlers
  - onclick, onchange, …

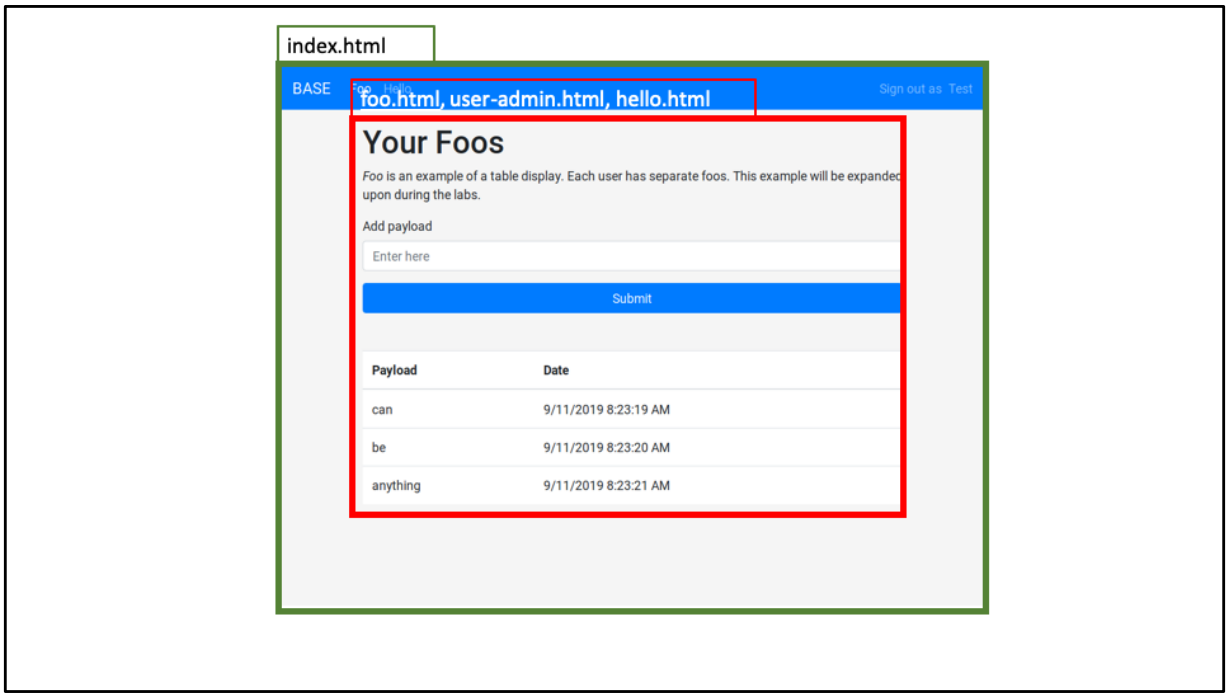
InnerHTML är smidigt sätt att ändra

Event handlers beror på vilket element man vill påverka

Notis: InnerHTML blir trevligare med ECMAScript 6 string templates `<a hre=${url}>Link</a>`

BASE tour


BASE
Foo
Hello
Sign out as Test
Your Foos
Foo is an example of a table display. Each user has separate foos. This example will be expanded upon during the labs.
Add payload
Enter here
Submit
Payload
Date
can
9/11/2019 8:23:19 AM
be
9/11/2019 8:23:20 AM
anything
9/11/2019 8:23:21 AM



Index.html och index.js definierar header

Behöver inte ha med den I varje tab

- Foo, Hello (lab 1), User admin, osv.

Essentials from index.html

```
<a href="#/foo">Foo</a>

<section id="main-tab"></section>

<script src="foo/foo.js"></script>
<script src="index.js"></script>
<script>
    base.mainController.initOnLoad(); // defined in index.js
</script>
```

Index.html innehåller länkar som byter tab I gränssnittet
Det finns en HTML tag där innehållet på tabben renderas dynamiskt
Den laddar också in alla skript

Essentials from index.js

```
const routingTable = {
    'foo': {
        partial: 'foo/foo.html',
        controller: base.fooController
    }
}

const newTab = 'foo';

fetch(routingTable[newTab].partial)
    .then(function(response) { return response.text()})
    .then(function(html) {
        document.getElementById("main-tab").innerHTML = html;
        routingTable[newTab].controller().load();
    });
```

routingTable innehåller information om tabbar
När en länk I navigationen klickas så görs rest anropet med fetch
I det här fallet blir newTab foo
Man hämtar en partiell HTML sida och byter in main-tab med det innehållet
Sedan anropas controllern definierad I routingTable

Essentials from foo.html

```
<template id="foo-template">
    <div></div>
</template>
```

En template tag representerar dynamiskt innehåll
Egentligen table, men vi förenklar till div här

Essentials from foo.js (1/2)

```
base.fooController = function() {
    let model = [];
    const controller = {
        load: function() {
            base.rest.getFoos().then(function(foos) {
                model = foos.map((f) => new FooViewModel(f));
                const t = document.getElementById('foo-template');
                model.forEach(() => d.render(t));
            });
    };
    return controller;
};
```

Ny controller för varje tab byte, så att man inte råkar spara gammalt tillstånd
Model behövs inte nu när vi bara ska rendera
Model sparar alla Foos, så att dom kan användas för update/delete funktionalitet

Essentials from foo.js (2/2)

```
const FooViewModel = function(_foo) {
    this.foo = _foo;
    this.render = function(template) {
        const div = template.content.querySelector('div');
        div.textContent = this.foo.payload;
        // this creates a new copy of everything under template
        const clone = document.importNode(template.content, true);
        // add clone to DOM tree
        template.parentElement.appendChild(clone);
    };
};
```

```
<template id="foo-template">
    <div></div>
</template>
```

Repetition av foo.html I rutan
FooViewModel innehåller foos och har logik för hur den renderas till HTML