Applikationsutveckling med JavaScript

Mikael Svahnberg*

2024-04-02

1 Introduktion

- Mer om applikationsutveckling i JavaScript
- Skrivbordsapplikationer
- Klient-Server-applikationer
- Testning

2 Gränssnitt med Javascript

1. Textbaserat terminalgränssnitt
2. Grafiska gränssnitt
3. En *Klient-Server* -arkitektur
4. Troget sitt ursprung är såklart websidor det primära gränssnittet.
   - Klient == webläsaren, med javascript
   - Server == microservices/containers/node.js

3 Textbaserade gränssnitt

- Som tidigare:
  - Input
    - Parametrar på kommandoraden
    - Konfiguration i filer
    - Inmatning från terminalen
  - Output
    - Utskrift till terminalen
    - Utskrift till filer
- Men också
  - Presentera resultaten som en websida

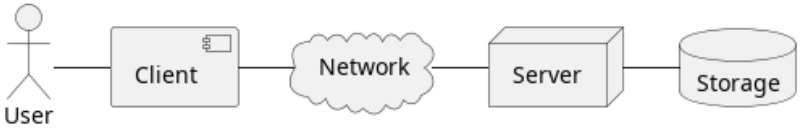
*Mikael.Svahnberg@bth.se

4 Grafiska gränssnitt

- De flesta grafiska gränssnitt för node.js baseras på en inbyggd-webläsare
  - Ofta Chromium eller Webkit
- Ramverk för att
  - skapa fönster
  - komma åt lokala resurser
- Använder vanlig html-kod för att generera själva GUI:t
- Exempel
  - Electron `https://electronjs.org/`
  - NW.JS `https://github.com/nwjs/nw.js`
  - NodeGUI `https://github.com/nodegui/nodegui`
- Allting packas ihop till en binär
  - Kan bli ganska stor (˜300 MB för en "tom" app).
- Svårt att köra från inuti en container
  - Dela upp applikationen i en GUI-del som körs på skrivbordet, och en microservice-del som körs som Containers.
  - ... Men varför använda JavaScript för GUI-delen då?

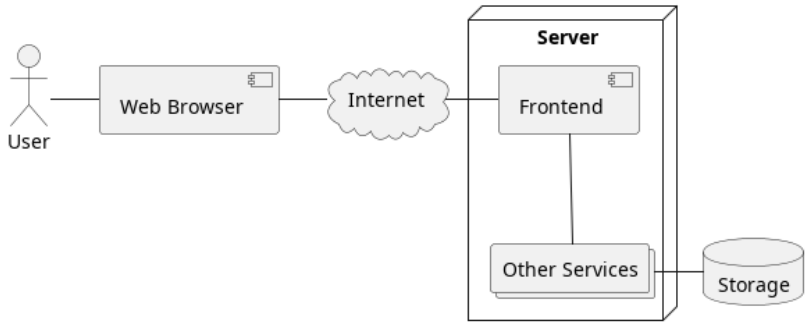
5 Klient-Server-arkitektur

- Klienter kommunicerar med Server med hjälp av något protokoll
  - Öppen standard, e.g. nntp, sendmail, http, irc, ...
  - Proprietär standard, e.g. zoom, discord, ...
- Kan vara flera olika sorters klienter (textbaserade, grafiska, ...)
- Var man lägger funktionaliteten varierar
  - Tung klient
    - Mer av funktionaliteten
    - Servern mer för lagring och kommunikation med andra
    - Öppna protokoll kräver mer misstänksamhet av Server-programvaran
  - Lättare klienter
    - "Dummare"; presenterar bara data och gör enkel inmatningskoll
    - Servern implementerar det mesta av funktionaliteten



6 Webapplikationer

- Ett slags Klient-Server - arkitektur
  - Webläsaren och Frontend delar på Klient-arbetet
- Webläsaren presenterar och gör enklare kontroll av inmatning
- Frontend
  - Väljer vad som skall presenteras
  - Sköter authenticering/kryptering mm.
  - Kontrollerar inmatning så den är ok
    - Kontrollerar även så att inmatning som redan kollats i webläsaren stämmer (eftersom vi inte kan lita på den)
  - *Statiskt* innehåll (statiska websidor, bilder mm.)
  - *Dynamiskt* innehåll (t.ex. data)



7 Skrivbordsapplikationer med Electron

- Om man nu *vill* skriva en skrivbords-javascript-applikation som inte körs i webläsaren.
- Electron är *ett* exempel - det finns några till.
- Baseras på *Chromium*
- Ramverk för att hantera fönster och resurser
- Händelsestyrt; man knyter ihop applikationen med EventListeners
- Själva gränssnittet byggs med HTML/CSS och JavaScript (client-side)
- Exempel:

```
const { app, BrowserWindow } = require('electron')

function createWindow() {
    const win = new BrowserWindow({ width: 800, height: 600 });
    win.loadFile('index.html');
```

```
}

app.whenReady()
    .then(() => { createWindow() });

app.on('window-all-closed', () => { if (process.platform !== 'darwin') app.quit() });
app.ipcMain.on('releasePony', (event, message) => { /* Find a Pony to release to pasture..
```

8 Electron API

- `https://www.electronjs.org/docs/latest/api/app`
- Exempel:
  - `app`
  - `BrowserWindow`
  - `clipboard`
  - `dialog` (t.ex. öppna-fil-dialoger)
  - `Menu`
  - `net`
  - `Notification`
  - `powerMonitor`
  - `screen`
  - `shell`
  - `TouchBar`
  - `Tray`

9 Utveckling, Test och Driftsättning

- Målet är att allt skall vara så lika som möjligt

  Utveckling lokalt eller i en Container

  Test först lokalt/container, sedan på en byggserver, sedan på en testmiljö
  - Successivt mer och mer lik driftmiljön
  - Antal noder/containers
  - Mängden data
  - Mängden nätverksfel/hårdvarufel, mm.

  Drift (eng. *Production environment*)
  - Ingen utveckling tillåten, bara kund-data.

- Det skall vara lätt (en kommandoradsparameter) att byta miljö
  - `make`, `make test`, `make testDeploy`, `make deploy`
- Tänk på accessrättigheter (vem får driftsätta produkten? måste de gå via `testDeploy`? osv.)

10 Vad skiljer mellan utveckling och drift

- debug-utskrifter
  - Använd `debug` (`const debug = require('debug')('debug-category')`) i stället för `console.log`
- loggar
  - `ENV NODE_ENV="production"` , bland annat för att inte skriva ut *stack-trace* vid fel
- Förmåga att starta om vid fel

  `nodemon` Startar om när filer ändras

  `forever` Startar om när programmet kraschar (Finns även andra alternativ som `StrongLoop` och `PM2`)

  Fundera på varför vill man inte använda `forever` när man utvecklar?
- **Säkerhet**, t.ex.
  - Accessrättigheter till databas
  - Filtreras kända sårbarheter bort? (ange en säkerhetspolicy till *Express*)
  - Avslöja inte mer än nödvändigt om vad du kör för drift-stack
    - `app.disable('x-powered-by');`
  - Begränsa IP-nummer om de gör ett visst antal felaktiga uppkopplingar
- Köra node.js i flera processer (`Cluster`)
  - Processer delar inte minne. Om detta är viktigt använd en extern tjänst (ex. *Redis*)
  - Lastbalansering, och vad händer med *Session Affinity* (en viss användare kopplar upp mot en viss nod)
- Förmåga att köra i bakgrunden (e.g. `docker --detach`)
- `minify` på websidor

11 Testning

- Det är en god vana att redan från början tänka på *unit-testning*
  - Manuellt när du arbetar med koden
  - Automatiserade tester för att ge dig trygghet
- Många testramverk använder *Behaviour-Driven Software Engineering*
  - Testet skall främst beskrivas med naturligt språk
    - ... och i slutändan kopplar man in den kod som faktiskt genomför testet.

- Ofta är fokus på att *köra* testerna
  - Själva utvärderingen inuti ett test lämnas till andra ramverk.

```
describe('Pinkie Pie Test', function() {
    it('should be able to say something', function() {
        let said = pinkiepie.speak();
        expect( said ).to.be.a.('string');
        expect( said ).to.not.equal('');
    });

    it('should only move left to right', function() {
        let [xPos, yPos] = pinkiepie.getPosition();
        pinkiepie.move();
        let [newXPos, newYPos] = pinkiepie.getPosition();

        assert.equal(newYPos, yPos, 'Pinkie Pie moved up or down');
        assert.notEqual(newXPos, xPos, 'Pinke Pie did not move left to right');
    });

    it('should stay on the screen', function() {
        pinkiepie.setPosition(0, 0);
        pinkiepie.move(-1, 0);
        let [newXPos, newYPos] = pinkiepie.getPosition();

        newXPos.should.be.at.least(0);
        newYPos.should.equal(0);
    });
});
```

12 Testning med Mocha

0. Mocha är ett av flera olika testramverk: `https://mochajs.org/`
1. Lägg till `mocha` i `package.json`:
   - Som en `depdendency` för att det skall installeras
   - Som ett `script` för att kunna köra det.
2. Skapa en katalog `./test` där du skriver testerna
   - God idé att skapa katalogstruktur och filer som härmar ditt projekt
3. `describe('det som testas', function() { ...Alla tester av X... });`
   - Kan ha flera `describe()` nästlade.
   - Exempel `Testing Pony \rightarrow Testing Movement`
4. `it('förväntat resultat', function() { ... utvärdering av det som testas ... });`

- it == Individual Test
- men också trevligt att sätta in i en mening "it should return X"
- Kan ha flera `it()` inom en `describe()`.

5. Använd *Hooks* för att ställa upp inför dina tester och städa efter:
   - `before('kort beskrivning', function() { ... })` körs en gång före alla tester i en `describe()`
   - `beforeEach()` körs före varje `it()`-test i en `describe()`
   - `afterEach()` körs efter varje `it()`-test i en `describe()`
   - `after()` körs en gång efter alla tester i en `describe()`

Notera:

1. Mocha kan fungera som nodemon: `mocha -w` kör testerna igen när filer ändras.
2. **Test** $\neq$ **Testkörning** Du behöver både arbeta med enhetstester och med att faktiskt köra programmet ibland.
3. Mocha kan bara testa funktioner/klasser som exporteras från moduler
   - dvs. Bara de som står med i `module.exports`
   - En testfil börjar typiskt att importera (`require` eller `import`) det som skall testas:
     - `const ElementUnderTest = require('../src/ElementUnderTest.js');`

13 Assertions

- Varje enskilt test gör en *begränsad* utvärdering av en funktion/ett objekt.
  - Testet utför en enskild handling, och utvärderar resultatet.
- Utvärderingen görs med *assertions*.
- Olika assertion-ramverk, till exempel:
  - Inbyggt i node.js `const assert = require('node:assert');`
  - should.js `https://shouldjs.github.io/`
  - expect.js `https://github.com/LearnBoost/expect.js`
  - Chai.js `https://www.chaijs.com/`
    - Stödjer *assert*, *expect*, och *should*.
- Mest olika "smaker" för hur man skriver testerna:

```
// Assert
assert( shouldReturnTrue(), 'It did not return true');
assert.equal( myName, 'Mikael', 'Names should match');
assert.equal( returnNumber(), 42, 'Number should be 42');
```

```
// Should
myName.should.be.a('string');
myName.should.equal('Mikael');
myName.should.have.lengthOf(6);

// Expect
expect(myName).to.be.a('string');
expect( returnNumber() ).to.equal(42);
```

14 Testa REST-API med Chai-HTTP

- Chai har också ett stort antal *Plugins*: `https://www.chaijs.com/plugins`
- Däribland `chai-http`
  - Lägger till metoden `chai.request()` i chai.
    - `chai.request(express-app)` öppnar `app`, genomför anropet, och stänger
    - `chai.request('http://localhost:80080)` genomför anropet mot en extern webserver.

Exempel på vad man kan göra:

```
chai.request(baseURL)
    .post('/form/submit')
    .set('Extra-HTTP-header', 'value')
    .type('form')
    .attach('uploadField', fs.readFileSync('testImage.png'), 'testImage.png')
    .send({ '_method': 'put', 'password': '123', 'confirmPassword': '123' ])
    .end( function(err, response) {
        // the usual assert/should/expect
    });

chai.request(baseURL)
    .get('/private')
    .auth(username, passsword)
    .query( {name:'Mikael', limitResults:10] )
    .end( /* ... */ );
```

15 Komplett exempel Mocha/chai/chai-http

```
let chai = require('chai');
let chaiHttp = require('chai-http');
let expect = chai.expect;
chai.use(chaiHttp);

let server = require('../src/app.js'); // app.js måste exportera den konfigurerade express

describe('Start Page', function() {
```

```
    it('should return a start page', function(done) { // done är en funktion som skall anr
        chai.request(server)
            .get('/')
            .end( (err, response) => {
                expect(response).to.have.status(200);
                expect(response.body.length).to.not.be.equal(0);
                done();
            });
    });
});
```

16 Best practices för Enhetstester

- Enkla, välbeskrivna tester
- Isolerade tester som inte beror på varandra
- Använd verktyg för att se till att all kod täcks
- Arbeta aktivt med testerna; skriv dem tillsammans med din kod
- Regelbunden refactoring

17 Sammanfattning

- Gränssnitt
  - text, grafiska, webapplikationer
  - andra client-server-arkitekturer
- Utveckling i en container eller skrivbordsapplikationer
  - Express/REST-API
  - Electron eller motsvarande
- Enhetstestning med Mocha och Chai

18 Nästa Föreläsning

- Introduktion till Persistent Lagring av Data och Databaser
  - Relationsdatabaser
  - NoSQL-databaser
  - Andra typer av databaser

Två resurser för en översikt om databaser och vad SQL är för något:

- `https://www.sql-easy.com/learn/what-is-a-database/`
- `https://www.programiz.com/sql/database-introduction`

19 Övning: Craic – ett enkelt chat-program

19.1 Introduktion

- I den här uppgiften skall vi arbeta med ett enkelt chat-program: *Craic*.
  - *Craic* är ett irländskt ord för skvaller.
  - Applikationen går ut på att man skriver små korta meddelanden till varandra.
  - `https://codeberg.org/mickesv/craic.git`

19.2 Utmaningar

Sätta sig in i en existerande kodbas
- Hellre än att jag går igenom systemet och systemarkitekturen så får *ni* glädjen att göra det.
- Hur kör man programmet?
- Vilka huvudsakliga komponenter finns?
- vad gör respektive modul?

Användargränssnitt från inuti en container
- Vi hade kunnat skriva en web-klient, men vi vill ha något annat
- Textbaserat UI, så kallat *TUI*

19.3 Kom igång med projektet

1. Ladda ner projektet: `https://codeberg.org/mickesv/craic.git`
2. Sätt dig in i projektet
   - Hur kör man programmet?
   - Vilka komponenter finns?
   - Vad gör respektive modul?
3. Testkör
   - Skriv några inlägg
   - Lägg datormusen upp-och-ner och försök använda bara tangentbordet
     - (tips: Man behöver trycka `<escape>` för att lämna ett textfält)
4. Kan du koppla upp dig mot en kollegas server? Hur?

19.4 Uppdatera Klienten

1. Lägg till ett textfält med namnet på servern som skall användas
2. Se till att den angivna servern faktiskt används
3. Testkör tillsammans med en kollega

19.5 Uppdatera Servern

1. Lägg till fler otillåtna ord och namn (notera att en del är angivna som *Reguljära Uttryck*)
2. Lägg till en modul som gör att man kan `#tagga` nyckelord och `@nämna` andra användare
   - Skall de bara sparas tillfälligt i servern eller skall de lagras i databasen?
   - Hur söker man efter en viss `#tag`? Lägg till det till serverns REST-API.
3. Lägg till stöd för att hämta flera sidor av inlägg
   - Man behöver lägga till `page=xxx` till frågan
   - Man behöver lägga till `page=xxx`, och `nextPage: yyy` i svaret.
   - Extrapoäng om ni inte använder sidnummer rakt av utan räknar fram en nyckel i stället.
     - (Att man kan lista ut sidnummer är ett säkerhetshål)

Fundera på:

- Vad händer om man använder ett förbjudet ord men stavar det annorlunda, t.ex. "belGIUm"?
- Hur kan du testa dina API-förändringar?
- Blir det en ny major version av produkten när du lägger till nya REST-ändpunkter?
- Blir det en ny major version när du lägger till stöd för att hämta fler sidor?
- Kan du förenkla servern så att den har en konfigurerbar lista med filter att tillämpa, snarare än att de är hårdkodade?
  - Fundera på hur du skulle implementera detta.

19.6 Skriv Tester

1. Planera och skriv Mocha/Chai-tester för servern
2. Planera och skriv Mocha/Chai-tester för klienten

19.7 Skapa en web-klient

1. Skapa en ny Container som kör en webklient på samma vis som den TUI-baserade klienten.
2. Kör programmet med både TUI-klienten och webklienten igång samtidigt.
3. Kan du återanvända dina tester från TUI-klienten?

19.8 Fundera på REST-API:et

- Hur vet klienterna om det finns nya meddelanden?
  - Vad innebär detta för servern?
- Hur kan du göra det annorlunda / snällare för servern?
  - Försök!

19.9 Sammanfattning

- Med containers och moduler blir varje del av programmet ganska fristående och lätt att anpassa.
- REST-API är inte lika enkelt som ett vanligt metodanrop, men nästan.
- Skalbarhet:
  - En egen container för att hantera `#taggar` och `@omnämnanden`?
  - Flera server-containrar med lastbalanserare?
- Olika typer av klienter