**Exam report – Introduction to Programming (T1)**

# Introduction

Our project has been to create a user-friendly web-shop, more specifically a web-shop that sells lamps. From the costumer’s perspective, two pages will be presented through the experience of shopping at coollamps.com: First a login/registration page followed by the actual shopping page where the user can browse through the product list before finally selecting the desired lamp that which then will appear in the cart with total price and quantity. An initial reflection, which admittedly is closer related business than programming, is if it intuitively makes sense to ask users to register and thus give away personal data to access the website. This would most likely make a significant portion of users leave the site before reaching the shopping page. Nevertheless, this project exam report will follow the logic of coollamps.com by examining the functions and though-processes behind firstly the login/registration page and then secondly the shopping page.

# Considerations and functions description of application

## Login/registration page

The initial idea concerning the login page was to manually type in the e-mail and password of each user manually in the HTML-file, which then would be the reference to the web-page on whether or not the user would be granted access to the shopping page. However, this idea was quickly discarded as this would result in a static function. The problem that arise from static coding is that it requires a lot of manual updating, which would continue to grow in complexity as the development of the application progresses. We therefore decided that the login and registration should follow a generic principle and that this mind-set ought to be followed throughout the project. Our first task was therefore to create a *class* called “User”, with a number of relevant *constructors* such as e-mail, name, password, zip-code and city (exhibit 1).

//Exhibit 1

class User {

constructor(email, fullName, streetName, zipCode, city, pw, repeatPw) {

this.email = email;

this.fullName = fullName; //More constructors outside exhibit 1

This built-up allows us to make a generic registration form that automatically saves the user-input in an array of users. As contrary to the static approach this means a user can login directly after having registered, easing the manual labour required from us developers. However, to avoid having to register each time one wants to expect the login function, we have three pre-defined users in the array.

The registration itself (Exhibit 2) functions from getting the registration form from the HTML-file via the getElementById() method. As its name suggest, it returns the element that matches the specified value, allowing us to manipulate it at give it the functions we deem necessary for a properly functioning registration page. We then begin a for-loop to save time and effort when checking if the user’s input meets the requirements that we shall later define. The loop runs through the registration form. If an element of the form, i.e. a text field, is left empty, an alert will pop-up stating that empty field needs to be given an input. If the input meets the requirements, the loop reaches its *return* *false* outcome which consequently stops the loop and progresses to the next element of the registration.

// Exhibit 2

document.getElementById("register").addEventListener("click",function(){

console.log('The Register button is clicked!');

var form = document.getElementById('registerForm');

for(var i=0; i < form.elements.length; i++){

if(form.elements[i].value === '' && form.elements[i].hasAttribute('required')){

alert('There are some required fields!');

return false;

}

}

Examples of functionality in registration form is to check is the e-mail provided actually constitutes an e-mail with ‘at’ and domain.

// Exhibit 3

function valEmail (mail) { //This will check if the input in the email fields match the requirements.

return /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)\*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/.test(mail)

A similar example is that the registration form checks if the zip-code provided is actually a Danish zip-code (Exhibit 4). The requirements for a Danish zip-code is that it contains four numbers and does not start with ‘0’. The script checks for this whilst checking if the registered e-mail is different (‘!==’) from the data in the existing user array as well as that the password provided matches the repeated password. Further, the registration form checks if the user actually inputs letters in the name fields (i.e. numbers are prohibited), that the user’s address is written in the form of street name (letters), a space and ended with a number, and finally that the password and repeat password field cannot be left empty.

// Exhibit 4

if (inputOnlyNumbers.test(registerZipCode) && registerZipCode.length == 4 && !registerZipCode.startsWith("0") &&

registerPassword === repeatPassword) {

}

else {

document.getElementById('registerResult').textContent = 'Your Zip code does not fulfill the requirements';

return false

It obviously constitutes a limitation to the application that data of a registered user is lost when the page is refreshed. This could be solved through local storage, but this has not been looked into for our project.

# 