A · P · I · I · T
Inspire love for learning

University of Staffordshire

Web Development and Operating Systems 1

JSON and Local Storage

Rangi Dahanayake

MSc. CS, BSc (Hons) Software Engineering

What is an API?

- Just like a Human–Computer Interface lets humans talk to computers (keyboard, mouse, touch),
- An API lets computers talk to other computers.
  - So instead of you clicking buttons, one program sends a request to another program, and gets a response.
  - An API allows computers and applications to communicate with each other.
- Used to request and exchange data between systems
- Most APIs return data in JSON format
- Commonly used in web and mobile applications
  - Payment Processing: E-commerce sites use APIs from providers like PayPal or Stripe to securely handle transactions and process payments without storing sensitive financial data themselves.

What is JSON?

- JSON stands for JavaScript Object Notation.
- It is a lightweight data format used to store and exchange data between computers.
- benefits:
  - human-readable
  - platform and language independent
  - lightweight

JSON Syntax

- JSON stores data as name : value pairs. **name** → the key (label) **value** → the data
  - "name": "Steve"
- Data items separated by commas
  - { "name": "Steve",
    "age": 21,
    "city": "Colombo" }
- `value` can be:
  - `string`
    - enclosed in double quotes → **"city": "Colombo"**
  - `object`
    - enclosed by braces
    - multiple `name` / `value` pairs

"student": {
"name": "Steve",
"age": 21
}

  - `array`
    - enclosed by square brackets
    - multiple `objects` → **"subjects": ["IT", "Maths", "English"]**

JSON example (1 of 2)

- how would you represent a phone number?
  - name of contact
  - type (mobile, home, work)
  - number

- this can be represented by an `object` containing three properties
  - each property has a `name` and a `value`:

```
{
    "name" : "Hero Gabler",
    "type" : "Mobile",
    "number" : "07912345678"
}
```

JSON example (2 of 2)

- how would you represent lots of phone numbers?
  - an `array` of phone number `objects`

```
[
   { "name" : "Hero Gabler", "type" : "Mobile", "number" : "07912345678" },
   { "name" : "Hero Gabler", "type" : "Home", "number" : "01234567890" },
   { "name" : "Zvi Ellery", "type" : "Mobile", "number" : "07912345678" },
   { "name" : "Vilma Sokal", "type" : "Work", "number" : "07912345678" }
]
```

Storing JSON in variables

- can store JSON object in a variable
  - can then be used in JavaScript

```
let directory =
[
   { "name" : "Hero Gabler", "type" : "Mobile", "number" : "07912345678" },
   { "name" : "Hero Gabler", "type" : "Home", "number" : "01234567890" },
   { "name" : "Zvi Ellery", "type" : "Mobile", "number" : "07912345678" },
   { "name" : "Vilma Sokal", "type" : "Work", "number" : "07912345678" }
]
```

Accessing JSON

- can access data by drilling down into structure
  - `[index]` to access array elements
  - dot notation to access fields

```
console.log(directory);
console.log(directory[0]);
console.log(directory[0].name);
```

Asynchronous activities

- JavaScript runs one line at a time, in order.
  - next line to be executed has to wait for current line to complete
- JavaScript uses one main thread in the browser.

That one worker (thread) must handle:

Painting the screen

→ showing text, buttons, animations

User actions

→ clicks, typing, scrolling

Reading data

→ files, API responses

Running your code

→ console.log, loops, calculations

But here's the key idea

It can only do ONE of these at a time.

Asynchronous activities

- Some tasks take time, like:
  - Reading a file
  - Fetching data from a server
- Waiting for a response
- If JavaScript does this synchronously:
  - The code stops and waits
  - Nothing else can run
  - Page freezes
- This is called blocking.

Asynchronous activities

- To fix blocking, JavaScript uses asynchronous operations.
- Asynchronous means:
  “Start this task, don’t wait, come back later when it’s done.”
- So:
  - File reading
  - API calls
  - Timers
- are done asynchronously, so the browser stays responsive.

Promises avoid waiting

- promises are used so that the thread is not tied up waiting for long activities to complete

- other code continues processing while promise code does its job

Promises in real life

- someone promises you that something will be done
  - although your plans may be deferred, you continue with every day life awaiting the outcome
  - once the outcome of the promise is known, it can be dealt with

- if a promise is **kept**
  - you can follow through with your plans
- if a promise is **broken**
  - you need to do something else
- if a promise is **not completed** within a certain time
  - you assume the promise has been **broken** and do something else

A real life example

- my friend has promised to get me a new phone
  - if everything goes to plan, they will **keep** their promise
  - if anything goes wrong, they will **break** their promise
  - if I don’t hear back from them after 6 weeks, I assume their promise is broken

- promise can be:
  - kept
  - broken
  - pending

- can only be kept or broken once
- cannot switch from kept to broken or vice versa

Promises in JavaScript

- we execute some code in a promise
  - the promise may succeed (**kept**)
  - the promise may fail (**broken**)

- we provide 2 functions
  - one that is executed if promise is successful (**kept**)
  - one that is executed if promise fails (**broken**)

Promises can be "chained"

- a successful promise can form part of a chain
  - if I get a phone, I promise to give my old phone to a different friend
  - but I can also break or keep my promise

- the same is true in JavaScript
  - the result of a promise may be passed to a new promise which could also be kept or broken

Fetching data from server

- fetch() API introduced in ES2015
  - simple way to implement asynchronous HTTPS (network) requests
  - uses promises
  - code only continues if promise is kept...
  - requires URL of the resource to fetch
  - provides built-in methods to convert the returned data:
    - text() to return data as text
    - json() to return data converted to json

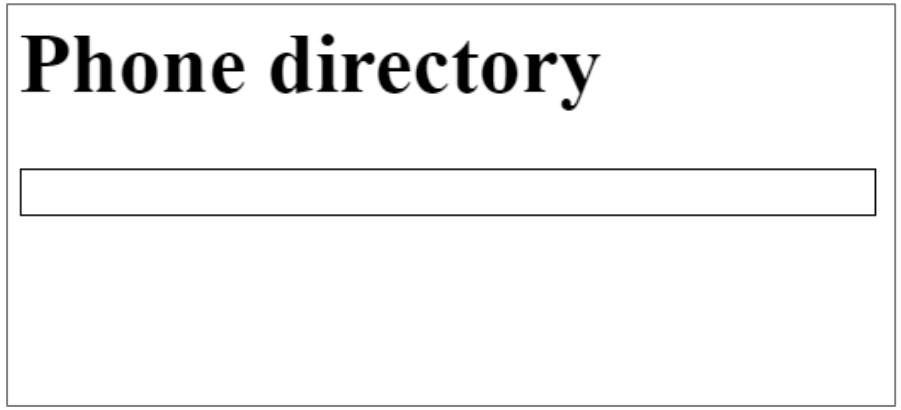
Worked example - directory

Directory

- an app is required to:
  - read an external JSON file
    - file contains names, types (mobile, home, work) and phone numbers
  - display file contents on screen when the page loads

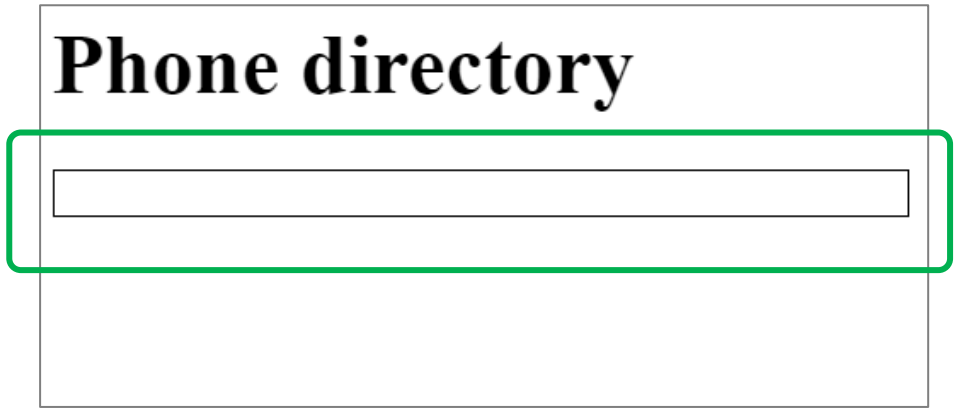
Step 1: sketch form

- heading labelled Phone directory
- area to display phone numbers



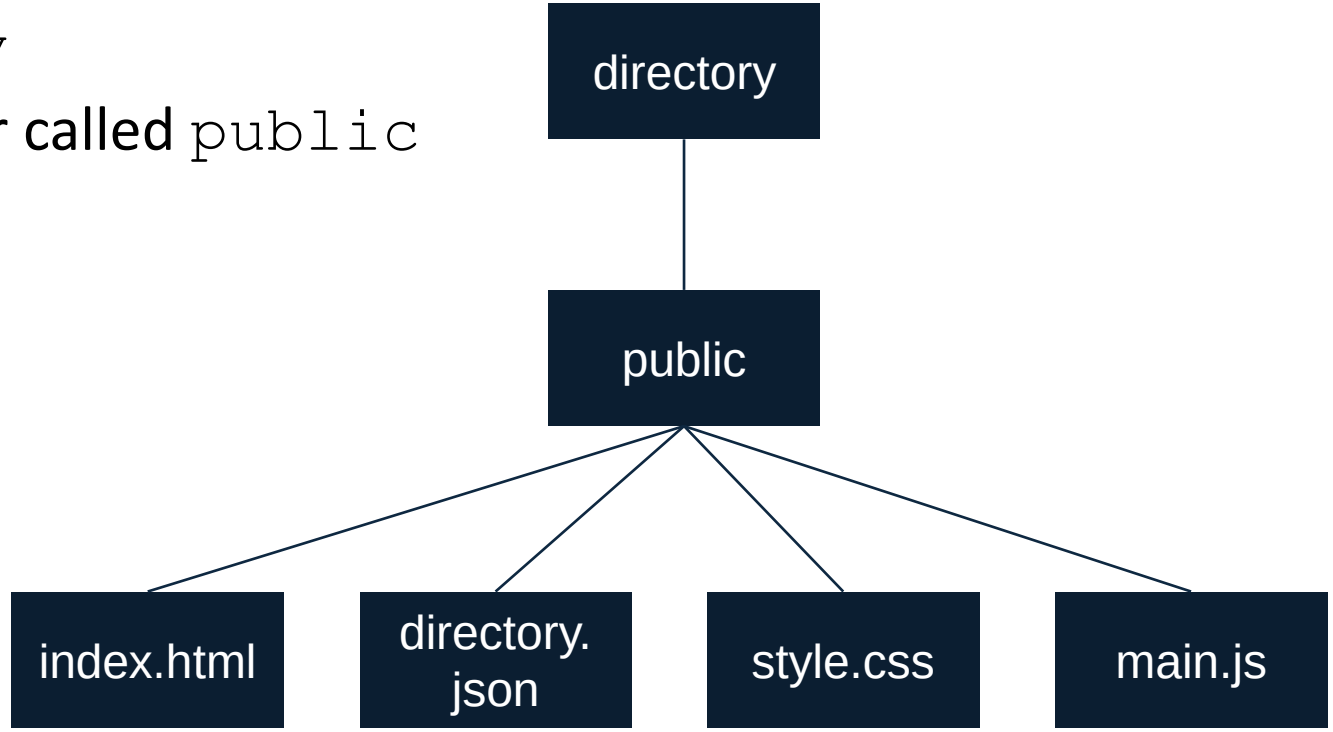
Step 2: decide what is interacted with

- program can:
  - output to div with `id=directory`



Step 3: set up site

- create a folder called `directory`
- inside `directory` create a folder called `public`
- inside `public` create:
  - `index.html`
  - `style.css`
  - `main.js`
  - `directory.json`



Step 3: create `directory.json`

```
[
    { "name" : "Hero Gabler", "type" : "Mobile", "number" : "07912345678" },
    { "name" : "Hero Gabler", "type" : "Home", "number" : "01234567890" },
    { "name" : "Zvi Ellery", "type" : "Mobile", "number" : "07912345678" },
    { "name" : "Vilma Sokal", "type" : "Work", "number" : "07912345678" }
]
```

directory.json

Step 4: create `index.html`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name=viewport content="width=device-width, initial-scale=1">
    <meta name="description" content="Fetching data using promises">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Phone directory</title>
    <link rel="stylesheet" href="style.css">
    <script src="main.js" defer></script>
</head>

<body>
    <h1>Phone directory</h1>
    <div id="directory"></div>
</body>

</html>
```

index.html

Step 5: create `style.css`

```
div {
    border: black solid 1px;
    min-height: 1em;
    width: 32em;
}
```

style.css

Step 6: decide what to do and when

```
//when page loads
    //get references to interactive elements
    //declare variables used by event handlers

    //fetch data from directory.json on server
    //convert retrieved data to json

    //loop through each entry
        //add entry to string
    //display string on page

    //if anything goes wrong
        //output error message
```

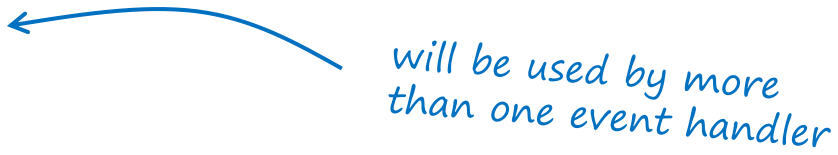
NB: this is known as an algorithm

Step 7: get references to interactive elements

```
//get references to interactive elements
const txtDirectory = document.getElementById("directory");

//declare variables used by event handlers
let directory;

getData();
```



main.js

Step 7: get data

main.js

```
function getData() {
    fetch("directory.json")
        .then(res => res.json() )
        .then(data => processData(data) )
        .catch(error => console.log(`Error - ${error}`)
    );
}
```

fetch data

if successful, then pass result of fetch to json()

if successful, then pass result of call to json() to processData()

if anything goes wrong output error

Step 7: test `processData()`

```
function processData(responseText) {
    directory = responseText;
    txtDirectory.innerText = "All data: " + JSON.stringify(directory);
    txtDirectory.innerText += "\n\nObject: " + JSON.stringify(directory[0]);
    txtDirectory.innerText += "\n\nField: " + (directory[0].name);
}
```

main.js

store response in directory

convert to string

already a string

Step 7: modify `processData()`


```
function processData(responseText) {
    directory = responseText;
    let dirList = "";
    //loop through each entry
    for (entry in directory) {
       //add entry to string
       dirList += `<p class='entry'>${directory[entry].name} (
           ${directory[entry].type} ) : ${directory[entry].number}</p>`;
    }
    //display string on page
    txtDirectory.innerHTML = dirList;
}
```

build a string containing formatted data

main.js

Step 8: testing

Test	Reason	Expected	Actual
load page	ensure data read from json file and correctly displayed	names, types and numbers output	

Web storage

Overview of web storage

- Web applications often need to remember information even after a page is refreshed or reopened.
- This avoids users having to enter the same data again and again, which improves usability.
  - Example:
    - Staying logged in
    - Remembering items in a shopping basket
    - Saving user preferences (theme, language)

Overview of web storage

- Cookies are the older, traditional way to store persistent data in web applications.
  - A cookie is a small piece of data (about 4 Kilobyte) stored in the user's browser
  - It is sent back and forth between the browser (client) and the server
  - Used to track user activity on a website
- Common uses:
  - Shopping cart contents
  - Login sessions
  - User preferences
  - Tracking visits and behaviour
- Because cookies are sent with every request, they can be slower and raise privacy concerns.

Overview of web storage

- The Web Storage API is a newer and simpler way to store data in the browser.
- In simple terms:
  - Data is stored only in the browser
  - Not automatically sent to the server
  - Can store much more data than cookies
  - Faster and easier to use with JavaScript

When you store data using the **Web Storage API** (like localStorage or sessionStorage), that data **stays only in the user's browser** and is **not included** in requests sent to the web server.

- There are two main types:
  - localStorage – data stays even after the browser is closed
  - sessionStorage – data is cleared when the browser tab is closed
- Common uses:
  - Saving form data
  - Remembering user settings
  - Storing temporary app data

Storage methods

- securely store `key-value` pairs
  - `keys` and their `values` must be **strings**
  - integer `keys` are automatically converted to strings

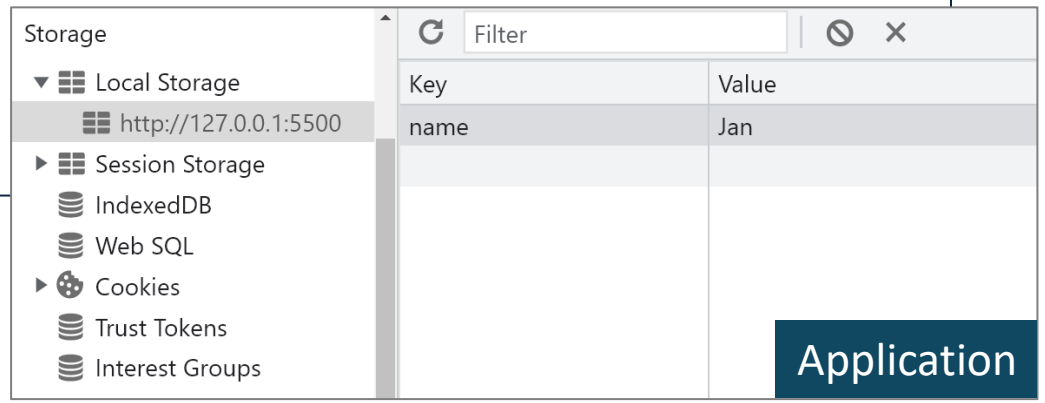
Method	Description
`setItem("key", "value")`	add `key` and `value` to local storage overwrite if `key` already exists
`getItem("key")`	retrieve a `value` using its `key`
`removeItem("key")`	remove an entry using its `key`
`clear()`	clear local storage

Check local storage

- local storage contents can be viewed using dev tools
  - **Console** window can be used to modify contents
  - **Application** window can be used to see contents

Console

```
localStorage;
localStorage.setItem("name", "Jan");
localStorage.setItem("name", "Bob");
localStorage.getItem("name");
localStorage.removeItem("name");
```



Application

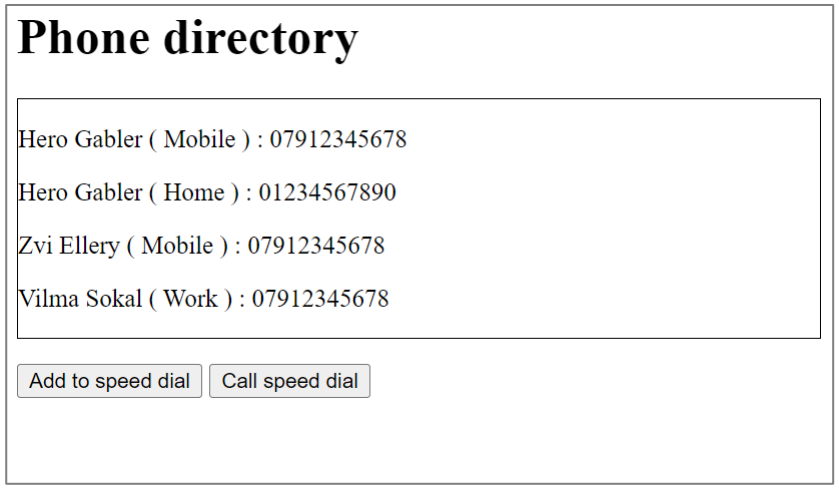
Worked example - speed dial

Speed dial

- the directory app is to be extended to:
  - highlight a directory entry when it is clicked
  - provide a button to save selected entry in local storage (replacing existing contents)
  - provide a button to display the contents of local storage

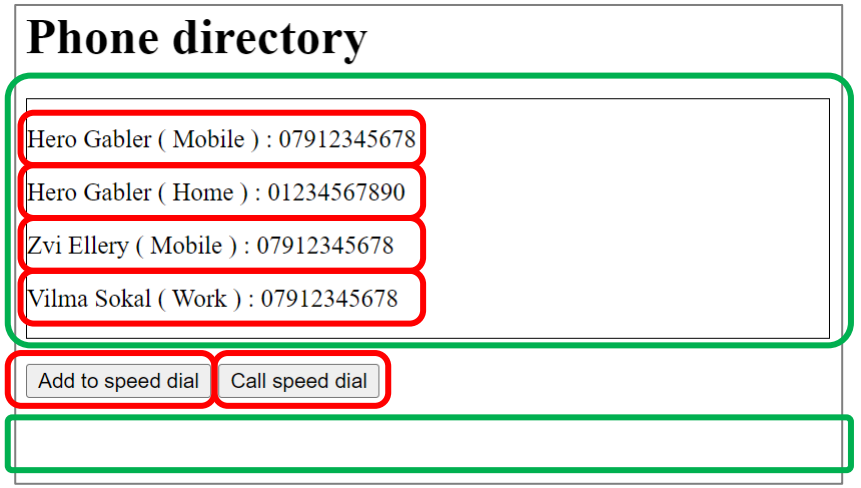
Step 1: sketch form

- heading labelled Phone directory
- area to display phone numbers
- button to add to speed dial
- button to call speed dial



Step 2: decide what is interacted with

- user can:
  - click any directory element with `class=entry`
  - click a button with `id=addToSpeedDial`
  - click a button with `id=callSpeedDial`
- program can:
  - output to div with `id=directory`
  - output to paragraph with `id=selected`



Step 3: set up site

- no changes required

Step 4: modify `index.html`

```
<body>
    <h1>Phone directory</h1>
    <div id="directory"></div>

    <p>
       <button type="submit" id="addToSpeedDial">Add to speed dial</button>
       <button type="submit" id="callSpeedDial">Call speed dial</button>
    </p>

    <p id="selected"></p>
</body>
```

index.html

Step 5: modify `style.css`

```
.highlight {
    background-color: lightyellow;
}

div {
    border: black solid 1px;
    min-height: 1em;
    width: 32em;
}
```

style.css

Step 6: extend what to do and when

- when directory entry clicked:
  - remove highlight from all entries
  - add highlight to selected entry
  - save contents of selected entry

- when add to speed dial button clicked:
  - add selected entry to local storage

- when call speed dial button clicked:
  - display data from local storage on page

NB: these are known as algorithms

Step 7: get references to interactive elements

```
//get references to interactive elements
const txtDirectory = document.getElementById("directory");
const txtSelected = document.getElementById("selected");
const btnAddToSpeedDiall = document.getElementById("addToSpeedDial");
const btnCallSpeedDiall = document.getElementById("callSpeedDial");

//declare variables used by event handlerslet directory;
let txtParas;
let selected;

//listen for events
btnAddToSpeedDiall.addEventListener("click", addToSpeedDial);
btnCallSpeedDiall.addEventListener("click", callSpeedDial);

getData();
```

main.js

Step 7: amend `processData()`

main.js

```
function processData(responseText) {
    localStorage.clear();
    directory = responseText;
    let dirList = "";
       for (entry in directory) {
           dirList += `<p class='entry'>${directory[entry].name} (
${directory[entry].type} ) : ${directory[entry].number}</p>`;
    }
    txtDirectory.innerHTML = dirList;

    txtParas = Array.from(document.getElementsByClassName("entry"));
    txtParas.forEach(item => item.addEventListener("click", displaySelected));
}
```

clear any existing values

getElementsByClassName returns a Collection which must be converted to an array

add click event listener to each element in array

Step 7: implement `displaySelected()`

main.js

```
function displaySelected() {
    for (let i = 0; i < txtParas.length; i++) {
        txtParas[i].classList.remove("highlight");
    }
    this.classList.add("highlight");
    selected = this.innerHTML;
}
```

remove highlighting from all elements then highlight selected one

save contents of selected element

Step 7: implement speed dial methods

main.js

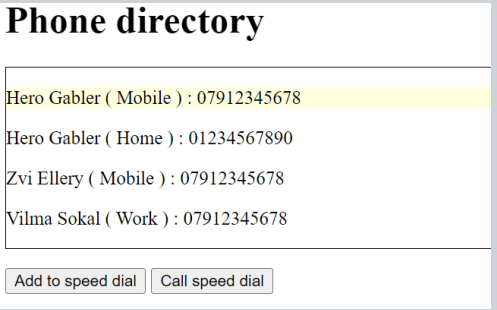
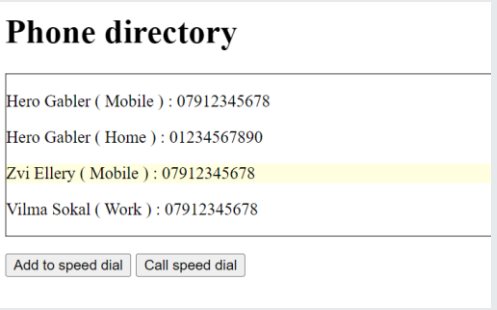
```
function addToSpeedDial() {
    localStorage.setItem("speedDial", selected);
}

function callSpeedDial() {
    txtSelected.innerHTML = localStorage.getItem("speedDial");
}
```

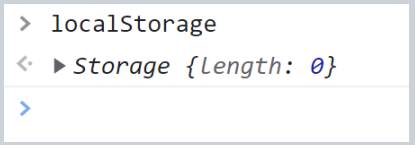
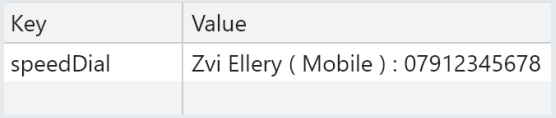
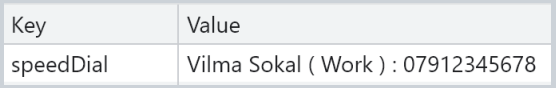
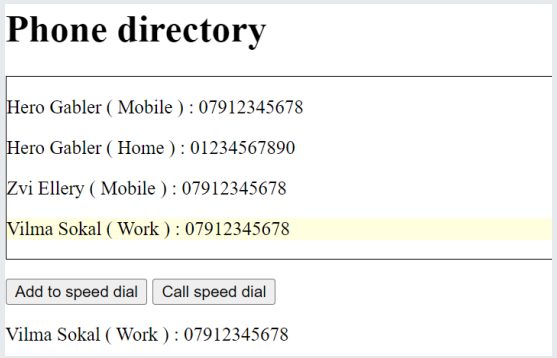
save selected item in local storage

retrieve value from local storage and display on page

Step 8: testing (1 of 2)

Test	Reason	Expected	Actual
click first entry (Hero Gabler)	check that selected entry highlighted	Hero Gabler highlighted	
click third entry (Zvi Ellery)	check that deselected entry highlight removed and highlight added to selected entry	Zvi Ellery highlighted	

Step 8: testing (2 of 2)

Test	Reason	Expected	Actual
type localStorage in Console window and use clear if it contains data	check that local storage empty	empty local storage	
select Zvi Ellery click Add to speed dial	check selected entry added to local storage	Zvi Ellery added	
select Vilma Sokal click Add to speed dial	check selected entry overwrites local storage	Vilma Sokal added	
click Call speed dial	check local storage entry retrieved and displayed	Vilma Sokal displayed	

Enabling and disabling buttons

Enabling and disabling

- any elements can be enabled or disabled from JavaScript:

```
//disable button
btnMyButton.disabled = true;
```

```
//enable button
btnMyButton.disabled = false;
```

- this can be based on a condition:

```
//disable button if count is 0
if (count == 0) {
    btnMyButton.disabled = true;
} else {
    btnMyButton.disabled = false;
}
```

or even better

```
btnMyButton.disabled = count == 0;
```