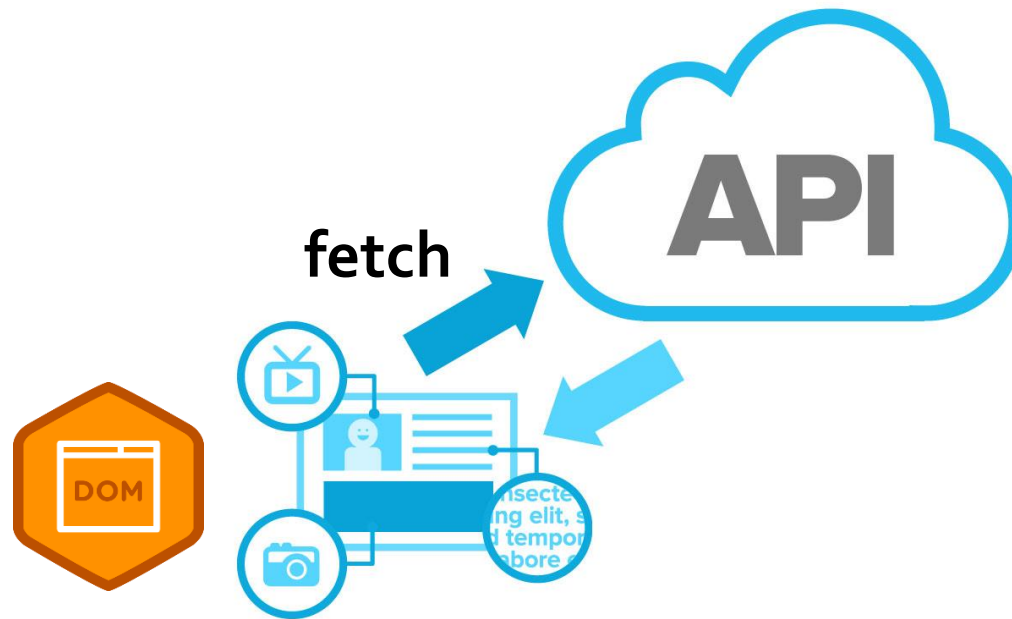


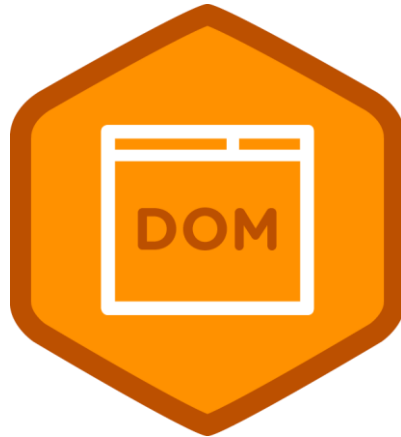
# JavaScript on the Client Side



# Outline

1. DOM Manipulation using JavaScript
2. Event Handling
3. Access Web API using Fetch
4. HTML Template to generate the UI
5. localStorage

# DOM Manipulation using JavaScript



# What can you use Client-side JavaScript for?

## Client-Side Dynamic Behavior

- **React to user input** i.e., handle client-side events such as button clicked event
- **Updating the page**
  - Add/update/delete page content: **Manipulate the Document Object Model (DOM)** of the page: read, modify, add, delete HTML elements
  - Change how things look: CSS updates
- **Validate form input** values before being submitted to the server
- **Perform computations**, sorting and animation
- **Perform asynchronous Web API calls** (AJAX) to get or submit JSON data to the server without reloading the page

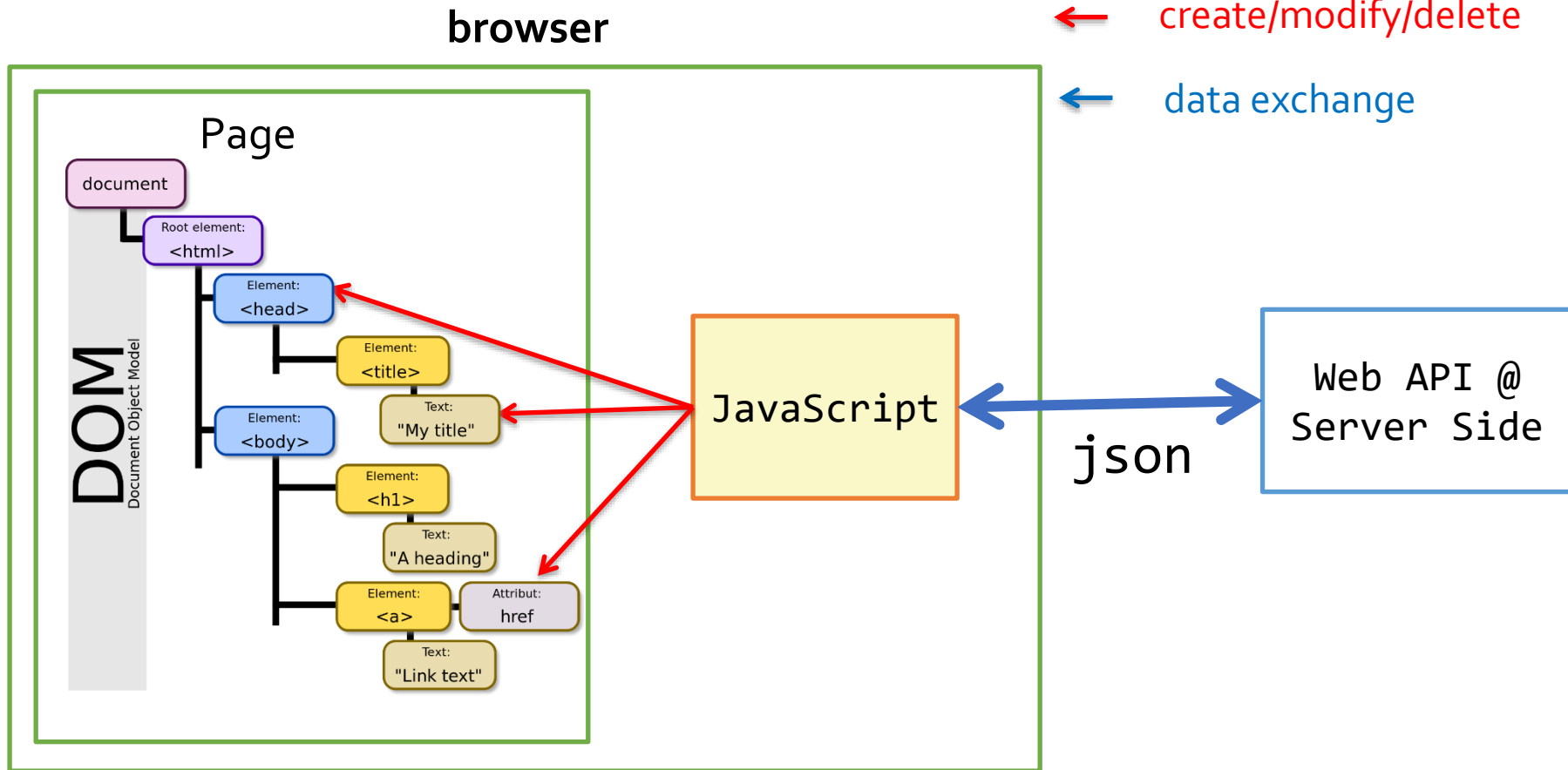
# Where to place JavaScript Code?

- The JavaScript code can be placed in:
  - `<script>` tag in the head
  - In an external file and add a reference to it in the HTML file. This is the recommended way
    - Reference via `<script>` tag in the **head** or at the end of the **body**

```
<script src="app.js"></script>
```

- JavaScript files usually have **.js** extension
- The **.js** files get cached by the browser

# Role of JavaScript on the Client Side

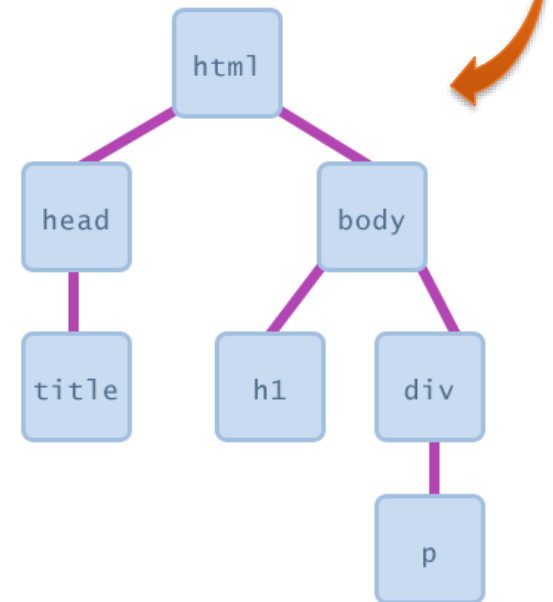


- DOM = A tree structure built out of the page HTML elements
- Use JavaScript to manipulate the DOM

# Document Object Model (DOM)

- DOM: Object-oriented Representation of the document
- DOM API consist of objects and methods to interact with the HTML page
  - **Select** page elements
  - **Add, update** or **remove** page elements
  - **Apply styles** dynamically
  - **Listen** to and **handle** events

```
<html>  
<head>  
  <title> ... </title>  
</head>  
<body>  
  <h1> ... </h1>  
  <div>  
    <p> ... </p>  
  </div>  
</body>  
</html>
```



# Example DOM Element

HTML

```
<p>
  Look at this octopus:
  
  Cute, huh?
</p>
```

DOM Element Object	
Property	Value
tagName	"IMG"
<u>src</u>	"octopus.jpg"
alt	"an octopus"
id	"icon01"

JavaScript

```
const img = document.querySelector('#icon01');
img.src = 'kitty.png';
```

document object  
represents the  
document displayed



# Selecting HTML Elements

- Elements must be **selected first** before changing them or listening to their events
  - **querySelector()** returns the first element that matches a specified *CSS selector* in the document
  - **querySelectorAll()** returns all elements in the document that matches a specified CSS selector

Example CSS selectors:

1. By tag name: `document.querySelector("p")`
  2. By id : `document.querySelector("#id")`
  3. By class: `document.querySelector(".classname")`
  4. By attribute: `document.querySelector("img[src='cat.png']")`
    - Return the first image whose src attribute is set to **cat.png**
- Examples
    - [https://www.w3schools.com/jsref/met\\_document\\_queryselector.asp](https://www.w3schools.com/jsref/met_document_queryselector.asp)
    - [https://www.w3schools.com/jsref/met\\_document\\_queryselectorall.asp](https://www.w3schools.com/jsref/met_document_queryselectorall.asp)

# Selecting Elements – old way!



- Access elements via their ID attribute

```
const element = document.getElementById("some-id")
```

- Via the **name** attribute

```
const elArray = document.getElementsByName("some-name")
```

- Via tag name

```
const imgTags = document.getElementsByTagName("img")
```

- Returns array of `<img>` elements

# DOM Manipulation

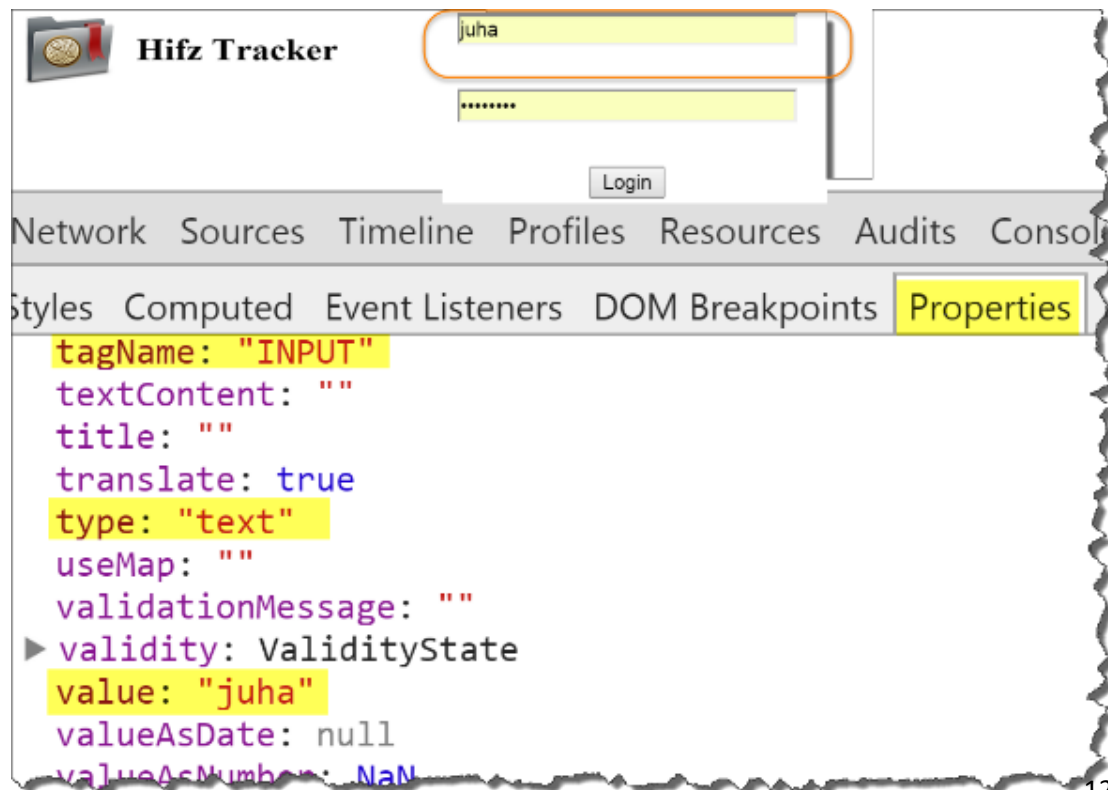
- Once we **select** an element, we can read / change its attributes

```
function change(state) {  
  const lampImg = document.querySelector("#lamp")  
  lampImg.src = `lamp_${state}.png`  
  const statusDiv =  
    document.querySelector("#statusDiv")  
  statusDiv.innerHTML = `The lamp is ${state}`  
}  
...  
 />
```

# Common Element Properties

- `value` - get/set value of input elements
- `innerHTML` - get/set the HTML content of an element
- `classList` - the `css` classes of an element

User Chrome  
Dev Tool to see  
the Properties of  
Page element



# Commonly used DOM methods

- **Add Element**

```
const newDiv = document.createElement('div')
newDiv.innerText = 'Div added by script'
document.body.append(newDiv)
```

- **Remove Element**

```
document.querySelector('#myDiv').remove()
```

- **DOM Traversal**

```
const parent = document.querySelector('#about').parentNode
const children = document.querySelector('#about').children
```

- **Hide & Show**

```
document.querySelector('#myDiv').style.display = 'none'
document.querySelector('#myDiv').style.display = ''
```

- **Add/Remove/Toogle CSS Classes**

- document.querySelector('#myDiv').classList.add('alert-success')
- document.querySelector('#myDiv').classList.remove('alert-success')
- document.querySelector('#myDiv').classList.toggle('alert-success')

# data attributes

- **data-\*** attributes allow us to store extra information on HTML elements
  - The name of a custom data attribute begins with **data-**
  - The name of a custom data attribute in JavaScript is the same HTML attribute but in camelCase and with no dashes, dots, etc.
- The **dataset** property provides read/write access to all the custom data attributes (data-\*) set on the element

# Dataset property

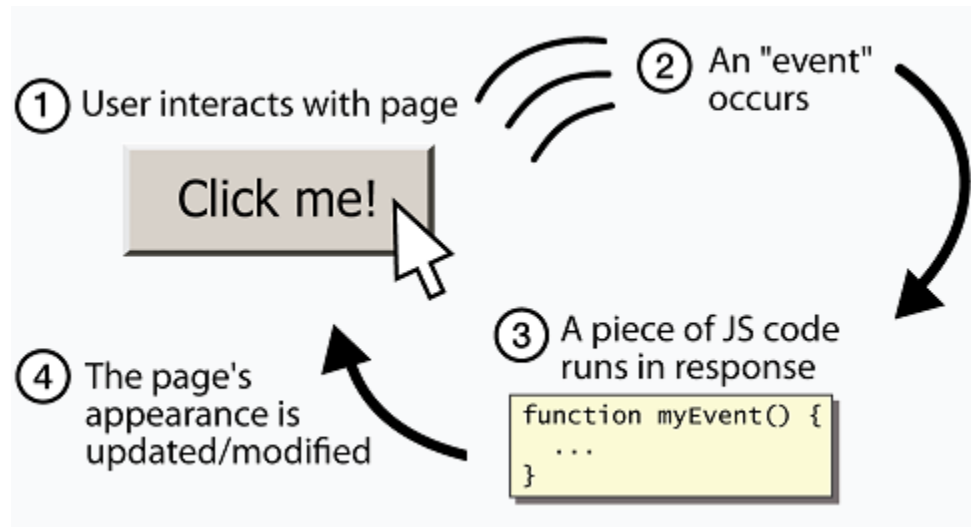
- Dataset property is used to read write custom data attributes set on the element

```
<div id="user"  
data-id="123456"  
data-user-name="johndoe">  
    John Doe  
</div>
```

```
const el = document.querySelector('#user');  
console.log(el.dataset);
```

```
// set a data attribute  
el.dataset.dob = '1960-10-03';  
console.log("dob: ", el.dataset.dob);  
  
delete el.dataset.dob;  
  
console.log("mobile: ", 'mobile' in el.dataset);  
if ('mobile' in el.dataset === false) {  
    el.dataset.mobile = '55751585';  
}  
console.log(el.dataset);
```

# Event Handling





# Event Driven Programming

- UI programming model is based on **event driven programming**
  - Code is executed upon activation of events
- An **event** is a signal from the Browser that some something of interest to the app has occurred
  - UI Events (click, change, drag)
  - Input focus (gained, lost)
  - Keyboard (key press, key release)
  - Page events (e.g., DOMContentLoaded)
- When an event is triggered, an event handler can run to respond to the event. e.g.,
  - When the button is clicked -> load the data from a Web API into a list

# Events Handling

- UI elements raise Events when the user interacts with them (such as a Clicked event is raised when a button is pressed)
- JavaScript can register event handlers to respond to UI events
  - Events are fired by the Browser and are sent to the specified JavaScript **event handler** function
  - Can be set with HTML attributes:

```

```



- Can be set through the DOM:

```
const img = document.querySelector("#myImage")  
img.addEventListener('click', imageClicked)
```

Ask to be notified of clicks on element **#myImage**

# Event Handler Example

```
<script>  
document.querySelector("#btnDate").  
    addEventListener("click", displayDate)  
  
function displayDate() {  
    document.querySelector("#date").innerHTML = Date()  
}  
</script>
```

Try it @

[http://www.w3schools.com/js/tryit.asp?filename=tryjs\\_addeventlistener\\_displaydate](http://www.w3schools.com/js/tryit.asp?filename=tryjs_addeventlistener_displaydate)

# Common DOM Events

- Mouse events:
  - **onclick**, onmousedown, onmouseup
  - onmouseover, onmouseout, onmousemove
- Key events:
  - **onchange**, onkeypress, onkeydown, onkeyup
  - Only for input fields
- Interface events:
  - onblur, onfocus, onscroll
- Form events
  - **onsubmit**: allows you to cancel a form submission if some input fields are invalid

# DOMContentLoaded

- DOMContentLoaded is fired when the DOM tree is built, but external resources like images and stylesheets may be not yet loaded
  - Best event for adding event listeners to page elements

```
//When the document is loaded in the browser then listen to studentsDD on change event  
document.addEventListener("DOMContentLoaded", () => {  
    console.log("js-DOM fully loaded and parsed");  
    document.querySelector('#studentsDD').addEventListener("change", onStudentChange)  
})
```

# The Event Object

```
function name (event) {  
    // an event handler function...  
}
```

- Event handlers can accept an optional parameter to represent the event that is occurring
- Event objects have the following properties/methods:

Property	Description
type	what kind of event, such as "click" or "mousedown"
target	the element on which the event occurred
timestamp	when the event occurred

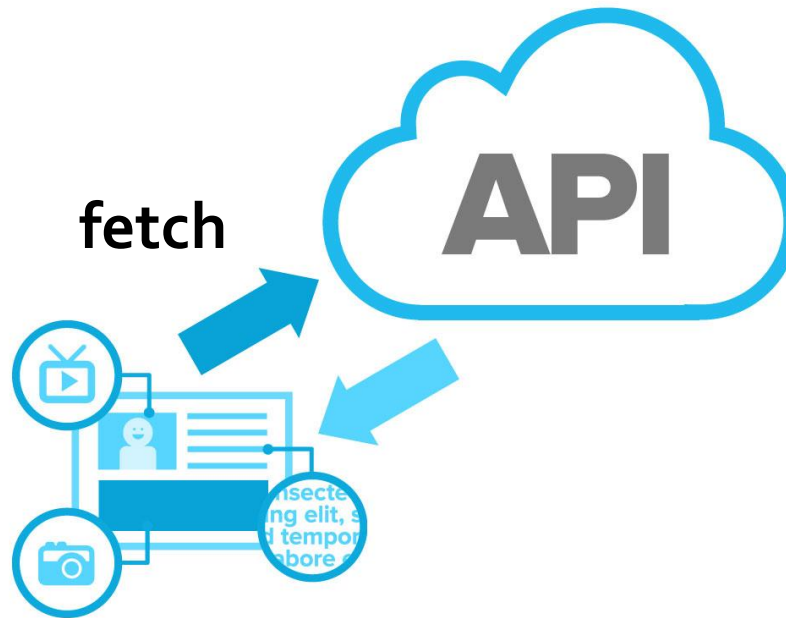
# Stopping an Event

- [preventDefault\(\)](#) stops the browser from doing its default action on an event.
  - for example, stops the browser from following a link when `<a>` tag is clicked
  - Or return false in an event handler to stop an event

```
<a href="#" onclick="onAddHero(event)">Add Hero</a>
```

```
async function onAddHero(event) {  
  event.preventDefault();  
  
  const heroesDiv = document.querySelector("#heroes");  
  const heroEditor = await getHeroEditor();  
  heroesDiv.innerHTML = heroEditor;  
}
```

# Access Web API using Fetch







- **AJAX** is acronym of Asynchronous JavaScript and ~~XML~~ JSON
  - AJAX is used for **asynchronously** fetching (in the background) of dynamic Web content and data from Web API
  - Allows dynamically adding elements into the DOM
- Two styles of using AJAX for partial page update
  - Load an HTML fragment and inject it in the page
  - Call Web API then use the received JSON object to update the page at the client-side using JavaScript

# Web API Get Request using Fetch

- Fetch content from the server

```
async function getStudent(studentId) {  
    const url = `/api/students/${studentId}`  
    const response = await fetch(url)  
    return await response.json()  
}
```

- **.json()** method is used to get the response body as a JSON object

# Web API Post Request using Fetch

- Fetch could be used to post a request to the server

```
const email = document.querySelector( "#email" ).value,  
    password = document.querySelector("#password").value
```

```
fetch( "/login", {  
    method: "post",  
    body: JSON.stringify({  
        email,  
        password  
    })  
})
```

`\${...}`

## HTML Template to generate the UI



# HTML template

- **HTML template:** a piece of HTML text that has some parts to fill in (placeholders)
  - The placeholders are filled with data from objects, the rest remains always the same
  - HTML template has **static parts** and **dynamic parts** (the gaps to fill in)

Date: \_\_\_\_/\_\_\_\_/\_\_\_\_  
Received from: \_\_\_\_\_, the amount of QR \_\_\_\_\_  
For: \_\_\_\_\_  
Received by: \_\_\_\_\_

- This template can be printed and used many times filling in the blanks with the data of each payment.
- **Template literals** could be used to define an HTML template to generate the UI.

# HTML template example

```
const payment = {
  date: '1/2/2021',
  name: 'Mr Bean',
  amount: 200,
  reason: 'Donation',
  receiver: 'Juha'
}

const receiptTemplate = (payment) =>
`<div>
  <p>Date: ${payment.date}</p>
  <p>Received from: ${payment.name}, the amount of QR ${payment.amount}</p>
  <p>For: ${payment.reason}</p>
  <p>Received by: ${payment.receiver}</p>
</div>`

console.log(receiptTemplate(payment));

// Render the template in the DOM
document.body.innerHTML = receiptTemplate(payment);
```

# Template Expressions

- **Expression interpolation:** a template literal can contain placeholders `${expression}` that get evaluated to produce a string value

```
const a = 5, b = 10;  
console.log(`${a} + ${b} = ${a + b}`);
```

- **Conditional expression**

```
const isHappy = true;  
const state = `${isHappy ? '😊' : '😞'} `;  
console.log(state);
```

# Display an Array using a Template literal

- Display an array elements using a template literal with the .map function

```
const days = ["Mon", "Tue", "Wed", "Thurs", "Fri", "Sat", "Sun"];
const daysHtml = `<ul>
  ${days.map(day => `<li>${day}</li>`).join('\n')}
</ul>`;
console.log(daysHtml);
```



# HTML template – Example 2

- Using HTML template to generate the UI

```
const person = {
  name: 'Mr Bean',
  job: 'Comedian',
  hobbies: ['Make people laugh', 'Do silly things', 'Visit interesting places']
}

function personTemplate({name, hobbies, job}){
  return `
```



# Local Storage

# Local Storage

- Local Storage provides mechanisms to **store key/value pairs** locally within the user's browser
  - The storage limit is at least 5MB and information is never transferred to the server
  - Web storage is per origin (per domain and protocol). All pages, from one origin, can store and access the same data

```
// Store
localStorage.lastname = "Saleh"
// Retrieve
console.log( localStorage.lastname )
// Delete
delete localStorage.lastname
```

## Note:

Name/value pairs are always stored as strings. Remember to convert them to desired format!

# localStorage Example

- Store the number of times a user has clicked a button
  - clickCount is converted to a number to be able to increase the counter

```
function clickCounter() {  
  if (localStorage.clickCount) {  
    localStorage.clickCount = parseInt(localStorage.clickCount) + 1;  
  } else {  
    localStorage.clickCount = 1;  
  }  
  document.querySelector("#count").innerHTML = `Button clicked  
    ${localStorage.clickCount} times.`;  
}
```

# Resources

- DOM

[https://developer.mozilla.org/en-US/docs/Web/API/Document Object Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

- Fetch API

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

- Web Storage API

[https://developer.mozilla.org/en-US/docs/Web/API/Web Storage API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API)