

Web-basierte Systeme

05: Browser Schnittstellen

Wintersemester 2024

Rüdiger Kapitza



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



Friedrich-Alexander-Universität
Technische Fakultät

Browser Schnittstellen

Vorlesungsplan

- 18. Oktober** Einführung und Darstellung von Webseiten (HTML und CSS)
- 8. November** Hypertext Transfer Protocol
- 15. November** Browser Schnittstellen
- 22. November** Kommunikationsschnittstellen im Browser
- 6. Dezember** WebAssembly
- 20. Dezember** **Architektur moderner Browser**
und Vorbereitung Papieranalyse
- 10. Januar** Clientseitige Architekturmuster und serverseitige
Implementierung von Web-basierten Systemen
- 17. Januar** Papieranalyse
- 24. Januar** Caching bzw.
Lastverteilung durch Zwischenspeicher
- 31. Januar** Aspekte von Web Sicherheit
- 7. Februar** Zusammenfassung und Ausblick

Zielsetzung der Lerneinheit

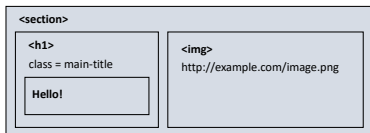
- Verständnis dafür wie innerhalb eines Browsers eine Webseite dynamische erzeugt und modifiziert werden kann
- Basiswissen Cookies und anderen Mechanismen zur Datenablage auf Seite des Browsers
- Grundlagen wie Webanwendungen zwischen Browser und Server kommunizieren können

Document Object Model

Document Object Model

JavaScript-Schnittstelle zur Manipulation von HTML-Dokumenten

- HTML-Seiten werden als baumartige Struktur aus JavaScript-Objekten repräsentiert
 - Document Object Model (DOM)
- Mittels JavaScript kann man in HTML-Dokumenten suchen und sie modifizieren
- Zugriff erfolgt über das global verfügbare `window.document`



Dokument Struktur

- Die DOM-Struktur folgt dem Aufbau eines HTML-Dokuments
 - `window.document.head`
 - `window.document.body`
- DOM-Objekte haben eine Vielzahl von Eigenschaften (~250)
- Diese Objekte haben eine gemeinsame Menge von Eigenschaften und Funktionen und werden als DOM-Knoten bezeichnet

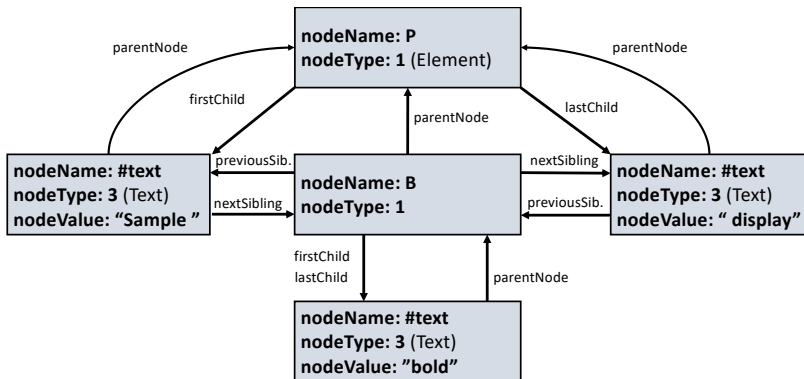
Dokument Struktur

- Identifikation von Elementen
 - nodeName – Attribut bestimmt den Typ z.B. P, DIV oder #text
- Es gibt Hilfen zur Navigation
 - Bspw.: parentNode, nextSibling, previousSibling, firstChild, lastChild
- Methoden um Attribute zu lesen und zu modifizieren
 - Z.B.: getAttribute und setAttribute usw.

Document Object Model

Navigation im DOM

```
1 <p>Sample <b>bold</b> display</p>
```



Navigation im DOM

- Man kann durch den DOM Schritt für Schritt navigieren ...

```
1 element =  
    document.body.firstChild.nextSibling.firstChild;  
    element.setAttribute(...
```

- Besser ist aber ein direkter Zugriff bspw. über ids:

- HTML: `<div id=div42>...</div>`

```
1 element = document.getElementById("div42");  
    element.setAttribute(...
```

- Oder mittels `getElementsByClassName()` für CCS Klassen, `getElementsByTagName()`, usw.

```
1 document.body.firstChild.querySelectorAll("p")
```

Weitere Standardmethoden des DOM

- `textContent` - Text eines Knotens und der untergeordneten Knoten
 - Für das P-Tag wäre dies: `"Sample bold display"`
- `innerHTML` - HTML der untergeordneten Knoten
 - `"Sample bold display"`
- `outerHTML` - HTML der untergeordneten Knoten, sowie das aktuelle Element
 - `"<p>Sample bold display</p>"`
- `getAttribute()/setAttribute()` - Lesen und setzen eines Attributes

Veränderungen am DOM

- Ändern des Inhalts eines Elements
 - `element.innerHTML = "This text is <i>important</i>";`
 - Verändert den Inhalt des Tags aber erhält die Attribute
- Ändern des Textinhalts eines Elements
 - `element.textContent = "This text is important";`
 - Verändert den Inhalt des Tags aber nur als Text
- Verändern eines Attributes z.B. `src` eines `img`-Tag:
 - `img.src="newImage.jpg";`

Zusammenspiel zwischen DOM und CSS

- Man kann beispielsweise die CSS class eines Elements verändern
 - `element.className = "active";`
- Veränderung der Sichtbarkeit von Elementen
 - Unsichtbar: `element.style.display = "none";`
 - Sichtbar: `element.style.display = "";`

Effektiver als Elemente aus dem Dom zu entfernen und wieder einzufügen.

- Auch direkte Veränderungen von Style-Anweisungen sind möglich (wenn man das *wirklich* will ;-)
 - `element.style.backgroundColor = "#ff0000";`
- Auch Anfragen an den DOM mittels CSS-Selektoren sind unterstützt
 - `document.querySelector()` (depth-first) und `document.querySelectorAll()`

```
1 var matches = document.querySelectorAll("p");
```

Verändern des DOM durch neue Knoten

- Erzeugen eines neuen Knotens
 - `element = document.createElement("P");`
 - `element = document.createTextNode("My Text");`
- Hinzufügen eines Knotens
 - `parent.appendChild(element);`
 - `parent.insertBefore(element, sibling);`
- Alternative: Klonen eines existierenden Knotens via `cloneNode()`
- Entfernen eines Knotens: `node.removeChild(oldNode);`

Weitere Funktionen

- Umleitung auf eine neue Seite
 - `window.location.href = "newPage.html";`
 - Kann zum unmittelbaren Wechsel & Abbruch des Skripts führen
- Einfache Wege um aus einem Skript heraus mit dem Benutzer zu interagieren
 - `console.log("Reached point A");`
 - `alert("Wow!"); confirm("OK?"); // Popup dialog`

Ereignisbehandlung

Interaktion zwischen DOM und JavaScript via Ereignisse

- Mausbewegung, Click, Betreten und Verlassen eines HTML-Elements
- Tastaturbezogene Ereignisse (drücken, loslassen und halten)
- Änderungen an Eingabefeldern und das Absenden eines Formulars
- Zeitbezogene Ereignisse
- Weitere Ereignisse
 - Inhalt eines Elements hat sich verändert
 - Seite oder Bilder sind erfolgreich geladen
 - Ausnahmefehler
- Hintergrundaktivitäten

Behandlung von Ereignissen

- Im Prinzip muss eine Behandlungsroutine drei Dinge ermitteln:
 1. Um welches Ereignis handelt es sich?
 2. Wo ist das Ereignis aufgetreten?
 3. Was ist nun zu tun?

- Variante 1: Direkt im Dokument

```
1 <div onclick="gotMouseClicked('id42'); gotMouse=true;">  
  ...</div>
```

- Variante 2: Über ein Skript verankert im DOM

```
1 element.onclick = mouseClicked;  
2 // or  
3 element.addEventListener("click", mouseClicked);
```

Behandlung von Ereignissen

- Ereignisbehandlungsroutinen erhalten ein Event-Objekt
 - Typischerweise abgeleitet – bspw. MouseEvent oder KeyboardEvent
- Basiseigenschaften eines Event 's
 - type - Name des Ereignisses
 - Bsp: 'click', 'mouseDown' und 'keyUP'
 - timeStamp - Zeitpunkt der Erzeugung des Events
 - currentTarget - Element an das die Ereignisbehandlungsroutine gebunden ist
 - target - Element welches das Ereignis ausgelöst hat

Behandlung von Ereignissen

- Das event wird als erster Parameter übergeben

```
1  function mouseClicked(event){  
2    console.log(event.timeStamp);  
3  }  
4  document.getElementById("mybutton").onclick=mouseClick;  
-
```

Maus- und Tastaturereignisse

■ MouseEvent

- button - Maustastendruck
- pageX und pageY - Position der Maus relativ zur oberen linken Ecke des Dokuments
- screenX und screenY - Position der Maus relativ zur oberen linken Ecke des Bildschirms

■ KeyboardEvent

- key - Unicode der gedrückten Taste

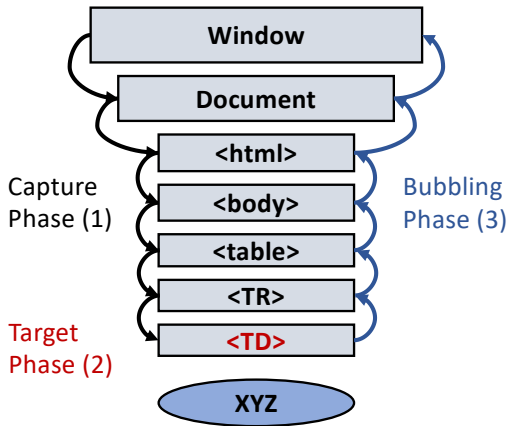
Aufrufabfolge von Ereignisbehandlungsroutinen

- Es gibt Situationen in denen Elemente überlappen
- Angenommen ein Benutzer klicket auf das "xyz "

```
1 <body>
2   <table>
3     <tr>
4       <td>xyz</td>
5     </tr>
6   </table>
7 </body>
```

- Wenn Ereignisbehandlungsroutinen für td, tr, table und body registriert sind, welche werden aufgerufen?
 - Manchmal soll nur das innerste Element den Aufruf annehmen...
 - ... manchmal wäre aber das Gegenteil sinnvoll.

Verarbeitungsablauf



Verarbeitungsablauf

- Capture phase
 - Es wird von dem obersten Element abgestiegen bis zum Auslöser
 - Zur Registrierung:
`element.addEventListener(eventType, handler, true);`
- Target phase
- Bubbling phase
 - Es wird von unten nach oben aufgestiegen
 - Zur Registrierung:
`element.addEventListener(eventType, handler, false);`
- Jede Ereignisbehandlungsroutinen kann auf dem Weg die weitere Behandlung unterbrechen:
`event.stopPropagation();`
- Die meisten Ereignisbehandlungsroutinen werden in aufsteigender Reihenfolge aufgerufen

Timer – nützlich für Animationen und regelmäßiges Aktualisieren

- Zeitverzögerter Aufruf einer Funktion (5s)
 - `token = setTimeout(myFunc, 5*1000);`
- Periodischer Aufruf
 - `token = setInterval(myfunc, 50);`
- Abbrechen eines Timers: `clearInterval(token);`

Nebenläufigkeit

- Ereignisse werden sequentiell verarbeitet (eins nach dem anderen)
- Ereignisse konkurrieren nicht mit anderen Skripten
 - *Run to completion*-Semantik und kein multi-threading!
- Relativ einfaches Model
- Hintergrundaktivitäten schwieriger zu realisieren
 - Lösung: Web workers

Web workers

- Web worker entspricht einem im Hintergrund ausgeführten Skript
- Im Vordergrund können weiterhin Ereignisse bearbeitet werden
- Code welcher per Web worker ausgeführt werden soll wird in einer separaten Datei abgelegt. Beispiel: `demo_workers.js`

```
1 var i = 0;
2 function timedCount() {
3     i = i + 1;
4     postMessage(i);
5     setTimeout("timedCount()",500);
6 }
7 timedCount();
```

Web workers

- `postMessage()` wird verwendet um Informationen zurück in die Webseite zu schicken

```
1   w = new Worker("demo_workers.js");  
2   w.onmessage = function(event){  
3     document.getElementById("result").innerHTML =  
       event.data;  
4   };
```

- Web workers werden separat ausgeführt und haben keinen Zugriff auf Objekte wie `window`, `document` oder das `parent`-Objekt
- `w.terminate()` zum beenden eines workers

Eigenheiten der Ereignisbasierten-Programmierung

- Code wird nur durch Ereignisse aufgerufen
- Behandlungsroutinen dürfen nicht lange laufen, da die Applikation währenddessen blockiert
- Feingranulare Programmierung durch viele kurze Behandlungsroutinen
- Als Ergänzung verwenden von Timern und Web workers
- Zusätzlich gibt es noch Service Worker welche als lokaler Proxy eingesetzt werden können

Ablegen von Daten im Browser

- Ursprünglich nur *Cookies* als Mechanismus verfügbar
 - Primärer Zweck ist die Wiedererkennung von Nutzern bzw. etablieren einer Sitzung
- Mittlerweile gibt es mehrere Alternativen zum Verwalten von Daten
 - *Web Storage* und seine Ausprägungen: *Local* und *Session Storage*
 - *Cache Storage*
 - Fokus auf Offline-Anwendungen und bisher noch eher experimentell im Kontext der *Service Worker*-Schnittstelle
 - *IndexedDB*¹
 - Transaktionale Datenbank welche JavaScript-basierte Objekte verwaltet
 - Flexibler und schneller Zugriff auf Größere Datenmengen

¹https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

Wiedererkennung von Nutzern

- HTTP-Anfragen beinhalten in Reinform kaum Informationen zur eindeutigen Identifikation eines Nutzer
 - Nutzer müssen aber im Rahmen von Sitzungen wiedererkannt werden
- Lösung zusätzliche Informationen die den Benutzer identifizieren
 - Information muss schwierig zu raten bzw. schwer fälschbar sein
- *Frühe* Lösung: Cookies
 - Information die von Webserver gesetzt wird und dann in jeder Anfrage des Nutzers enthalten ist

Weiter Verwendungszwecke von Cookies

- Personalisierung und Tracking (!)

Verwendung von Cookies

- Webserver können Cookies in ihre HTTP Antwort einfügen

```
1 HTTP/1.0 200 OK
2 Content-type: text/html
3 Set-Cookie: yummy_cookie=choco
4 Set-Cookie: tasty_cookie=strawberry
5
6 [page content]
```

- Diese Information wird nun in allen folgenden Anfragen des Nutzers mittels des Cookie-Header übernommen

```
1 GET /sample_page.html HTTP/1.1
2 Host: www.example.org
3 Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Rahmenbedingungen von Cookies

- Cookies werden nur innerhalb der erzeugenden Domäne vermittelt
 - Rechner, Portnummer und evtl. URL-Anteil
- Cookies haben ein typischerweise ein Verfallsdatum und können durch den Browser gelöscht werden
- Weitere Limitationen
 - Cookies können meist nicht größer werden als 4KB
 - Anzahl der zulässigen Cookies ist beschränkt (bspw. max. 50)

Cookies als browser-seitiger Speicher?

- Eher kritisch zu sehen!
- Benutzer haben volle Kontrolle über Cookies
 - D.h. modifizieren, löschen, erzeugen und verlieren von Cookies
- Nutzer sind auch eher skeptisch gegenüber Cookies eingestellt
 - Vielfache Verwendung von Cookies zum Tracking
 - *Do Not Track* header und entsprechende Hinweise auf Datenschutzrichtlinien sollen die Situation verbessern
- Fazit: Als Speicher für Daten auf Browserseite mittelmäßig geeignet aber sinnvoll/notwendig zur Etablierung von Sitzungen

Verwendung von Cookies zur Sitzungsverwaltung

- Frühe Rahmenwerke stellten eine einfache Schnittstelle bereit, um Sitzungsdaten als Cookies zu übermitteln
 - Bsp. auf Server-Seite: `session[:user_id] = "mendel"`
 - Abgelegt Information wird automatisch in alle folgenden Antworten/Anfragen eingefügt
 - Eingefügte Daten werden auf Server-Seite automatisch genutzt um Sitzungsdaten zu ermitteln
 - Falls kein Cookie gefunden wird – einfach neuen anlegen – entspricht einer neuen Sitzung
- Nach Abarbeitung einer Anfrage müssen jedoch die Sitzungsdaten abgelegt werden. (Wo?)

Sicherheit von Cookies

- Da Cookies in jeglicher Form auf Benutzerseite modifiziert werden können ist Vorsicht angebracht!
- Bspw. verschlüsseln der Daten
 - Vertraulichkeit und Integrität werden geschützt
 - Löschen immer noch möglich!
- Alternative nur eine Referenz als Cookie
 - Set-Cookie: `session=0x4127fd6a; Expires=Mon, 07 May 2018 11:18:14 GMT`
 - Weniger Transfer aber dennoch kryptographische Prüfsumme oder Ähnliches nötig

Sicherheit von Cookies

- Im Prinzip können Cookies auch im Browser durch Skripte erzeugt und modifiziert werden via `Document.cookie`

```
1 document.cookie = "yummy_cookie=choco";  
2 document.cookie = "tasty_cookie=strawberry";
```

- Nicht alle Cookies sind in Skripten zugänglich
 - Durch `HttpOnly` kann der Zugriff verwehrt werden
- Über `Secure` kann man erzwingen das Cookies nur über eine verschlüsselte Verbindung übertragen werden
- Cookies hatten und haben viele Probleme (Cross-site scripting (XSS) und Cross-site request forgery (CSRF))
- Tracking for allem über *third-party Cookies*

Web Storage

- `sessionStorage` – Sitzungsspeicher gekoppelt an die Laufzeit eines Fensters/Tabs
- `localStorage` – Langzeitspeicher per Domäne
 - Lagerung über Sitzungen hinweg
- Zugriff über `Window.sessionStorage` bzw. `Window.localStorage`
- Modifikation erfolgt über eine key/value Schnittstelle

```
1 localStorage.colorSetting = a4509b;  
2 localStorage['colorSetting'] = '#a4509b';  
3 localStorage.setItem('colorSetting', '#a4509b');
```

- Über Änderungen am Storage kann man sich via `StorageEvent` informieren lassen
- Speicherplatz ist typischerweise limitiert auf 10MB

Zusammenfassung

- Mittels der Schnittstellen des DOM können Webseiten sehr vielfältig verändert werden
- Ereignisbasierte Programmierung erfordert die feingranulare Aufteilung in einzelne Skripten
- Ereignisse werden sequentiell abgearbeitet
 - Vorteil: Einfach zu programmieren
 - Nachteil: Schlecht Ausnutzung von Mehrkernsystemen bzw. geringe Nebenläufigkeit
 - Alternative für rechenintensive Aufgaben: Web worker
- Daten können auf Browserseite abgelegt werden
 - Cookies als bisherige Standardlösung mit einer Reihe von Nachteilen
 - Mit Web storage und anderen Lösungen neue Möglichkeiten und sukzessive Ausrichtung auf komplexe Anwendungen
 - Auch wichtig für Offline-Anwendungen