Einführung und Übersicht; Webentwicklung

Ein gedanklicher Rahmen für den Einstieg
in “Interaktive Medien” für MMP an der FHGR.

1. Januar 2025

Inhaltsverzeichnis

Einführung

Dieses Dokument ist als gedanklicher Rahmen zum späteren Einordnen von diversem Stoff aus der Webentwicklung gedacht. Zudem soll die Brücke zum Anwenden geschlagen werden. Es wird praktisch kein Verständnis über Webtechnologien vorrausgesetzt. Somit wird gehofft, dass dieses Dokument vielleicht als Vorbereitung oder Einstiegsguide für "Interaktive Medien" für manche neue MMP Studenten funktionieren könnte. Es bestehen sicherlich Fehler oder Unvollständigkeiten. Wenn daran Interesse besteht, soll dieses Dokument gerne angepasst und weitergeführt werden. Die latex Quelldateien werden veröffentlicht.

Präambel Programmieren

In den entsprechenden, folgenden Kapiteln wird nicht versucht die gesamte Funktionalität oder Syntax von Javascript oder PHP zu umreissen. Stattdessen sollen bloss eine Auswahl an Eigenheiten, und für den Einstieg in die Webentwicklung wichtigen Funktionalitäten, betrachtet werden. Es sollten die typischen Strukturen[1] einer Programmiersprache bereits bekannt sein.

Fürs Verständnis von irgendetwas gegebenem in einer Programmiersprache, kann es oft helfen die Dinge die bei Ausführung[2] tatsächlich passieren, ein ganz kleines bisschen Hardware-näher zu erfassen. Denn man kann beim erlernen einer Sprache sich manchmal "in der Syntax selbst verfangen". Obwohl das was tatsächlich im Kontext zwischen dem Prozessor und dem Arbeitsspeicher passieren würde, gar nicht so schwer zu verstehen wäre, aber die vielen "coolen" Wörter und Beschreibungen auf dem Level der Syntax nicht immer direkt ineinandergreifen und aufeinander aufbauen. Ein solches herunterbrechen hilft beim erlernen einer Sprache wie C++ enorm. Da Javascript aber einiges an Abstraktion mehr besitzt, und dazu noch interpretiert ist, und sehr einfach mit dem Netzwerk interagiert werden kann, wird es schwerer auf ein ähnliches Level runter zu kommen. Trotzdem soll jenem Aspekt im Folgenden ein wenig Rechnung getragen werden, aber nur soweit dass der Praxisbezugs noch berücksichtigt wird.

[1] Entspricht etwa: Variablen, Datentypen, Operatoren, Kontrollstrukturen, Funktionen, Ein-/Ausgabe, Arrays/Datenstrukturen. Jenes Grundlagenwissen kann man gut in Kursen wie auf "Codeacademy" lernen.

[2] "during runtime".

1 Grundlegendes

Das Internet kann man sich als eine grosse Menge von Computern die irgendwie untereinander vernetzt sind, und kommunizieren können, vorstellen. Um Websiten zu verstehen, lohnt es sich dabei anzufangen, dass eine gegebene Website in ihrer Gesamtheit in irgendeiner Form auf einem Server-Computer liegt. Und einem Client-Computer auf Anfrage geschickt werden kann. Dabei ist **Client** und **Server** mehr eine Rollenbezeichnung während einer gegebenen Kommunikation. Jeder Computer könnte an sich, je nach Situation, beides sein.

Eine **URL**[3] wie bspw. die Zeichenfolge "www.google.com" entspricht auch einer für Menschen lesbaren Internetaddresse. Als ganzes entspricht sie einem bestimmten Server. Sie muss konventionell in drei Teile gegliedert werden; Die **Topleveldomain** ("com") ist am wenigsten spezifisch. Das heisst es gibt mehr URL's mit einer gegebenen Topleveldomain, als URL's mit einer gegeben Domain, oder Subdomain. Eine **Domain** (wie "google") muss meist für gegebene Zeiträume und in Kombination mit einer Topleveldomain bezahlt werden. Das bedeutet, dass man Kontrolle darüber erhält, welche IP-Adresse[4] der gekauften "Domain.Topleveldomain" zugeordnet wird. **Subdomains** (wie "www") können bei Verfügung über eine Domain meist beliebig addiert werden, und daher muss es nicht immer "www" vor einer Domain haben.

Wenn du eine URL in deinem Browser eingibst, schaut er nach, ob er die zugehörige IP-Adresse gecached hat. Falls nicht, schickt er die URL an einen DNS[5]-Server deines Internetproviders. Dieser übersetzt die URL in eine IP-Adresse und schickt sie an deinen Computer zurück[6]. Dein Client kann mit der IP-Adresse mit dem entsprechenden Server Kontakt aufnehmen, und könnte ganz verschiedene Dinge erfragen. In diesem Beispiel nehmen wir wohl an, dass es sich um die Quelldaten einer Website selbst handelt, aber es können auch Daten in anderen Formen sein, die ebenfalls wichtig für jene Website sein könnten.

Die meisten Websites bestehen aus drei Typen Quellcode. Das wäre **HTML**, **CSS** und **Javascript**. Quellcode ist meist einfach eine Textdatei[7]. Du kannst also um HTML, CSS oder Javascript zu erstellen einfach eine Textdatei mit '.html', '.css' oder '.js' kennzeichnen und losschreiben. Um jene Dateien auf einen Server zu kriegen, bietet ein Web-hosting Anbieter meistens einen FTP-Zugang auf den Server, bzw. das "Root-Verzeichnis". FTP ist ein Kommunikationsprotokoll derselben Ebene wie HTTP. (Siehe Abschnitt "Kommunikationsprotokolle") Es wird meist ein Benutzername und Passwort benötigt.

Die Quelldaten werden von deinem Browser interpretiert oder "geparsed[8]". Dazu geht er durch das HTML durch, denn im HTML ist das CSS und Javascript eingebunden[9], oder es wird darauf verwiesen. Man kann das sich so vorstellen, als ob alles, dort wo auf es verwiesen wird, in das HTML hineingeschrieben wird. Und dann setzt der Browser um, was ihm der Quellcode gibt. Somit liegt der Browser zwischen Quellcode und dessen Ausführung auf deinem Computer. Dadurch ist der Quellcode universeller, und schädliche Aktionen

[3] Universal Ressource Locator.

[4] Internet Protocol - Adresse: Eine für Menschen weniger lesbare Internetadresse Bspw.: 192.158.1.38.

[5] Domain Name System.

[6] Wenn er die IP zu einer gegebenen URL nicht gespeichert hat, erkundigt er sich zuerst über womöglich mehrere Hierarchiestufen bei anderen DNS-Servern. Aber das ist nicht wichtig für die Inhalte dieses Dokuments.

[7] Daten sind auf einem Computer natürlich immer binäre Zahlenabfolgen. Deren Sinn ergibt sich meist einfach daraus wie gewisse Programmierer bestimmte Daten für ihr Programm strukturieren wollten. Eine Photoshop Datei, gekennzeichnet durch ".psd", wäre eine solche Datei.
Mit Textdateien meint man aber Daten welche über eine bestimmte Standardisierung Textzeichen kodieren. Es werden also einfach einer bestimmten 8- oder 16-bit Zahl ein bestimmtes Zeichen zugeordnet. Der übliche Standard dazu wird UTF-8 genannt. "ASCII" ist älter und bekannter, aber UTF-8 ist darauf rückwärts kompatibel.

[8] Die Reihenfolge in welcher der Quellcode geparsed wird ist in der Praxis bloss für eins wichtig: Es kann sein, dass ein script nach einem Element des HTML's sucht. Wenn es aber vor jenem Element eingebunden ist, dann kann der Browser beim Parsen jenen Befehl noch nicht umsetzen. Um das zu verhindern, kann man zu einem script tag 'defer' dazu schreiben, was verursacht, dass es erst am Ende ausgeführt wird.

[9] Siehe "link" oder "script" HTML tag.

durch den Quellcode einer Website selbst sind nicht möglich.

Das bedeutet aber auch, dass es schwierig ist *genau* zu wissen was durch einen gegebenen Quellcode auf deinem Computer passieren wird. Anweisungen die ein Computer direkt ausführen kann, werden also in der Webentwicklung nie generiert. Programme wie bspw. der Browser selbst, oder Excel, entstanden daraus, dass Quellcode in Anweisungen die dein Prozessor versteht, übersetzt werden. Das bloss als Anmerkung um zu verdeutlichen, dass "interpretierter" Quellcode für Webseiten nicht gleich einzuordnen ist wie Programme aus "kompiliertem" Quellcode in einer Programmiersprache wie Rust oder C.

Wenn der Browser eine Website vollständig geparsed hat, stellt er sie dar. Dazu liefert das HTML dem Browser eine Menge von Einheiten, die aufeinander aufbauen. Die tatsächliche Darstellung jener Einheiten, sprich die bestimmten Grössen, Positionen und Weiteres, wird durch das CSS bestimmt. Bis hier hin wäre eine Website einem statischen Dokument noch sehr ähnlich.

Javascript wird aber im Browser, bzw. "JS-Engine", wie ein Programm ausgeführt. Über Javascript kann man auf die Seitenstruktur[10] zugreifen, und sie manipulieren. Somit kann sich die Website über die Zeit dynamisch verhalten. Zudem kann der Browser, und damit auch Javascript, wieder raus ins Internet gehen, verschiedenste Dinge tun, und wieder auf den Client holen[11]. Nun wurde also grob überspannt in welcher Form eine Website auf dem Client bzw. Browser ankommt, und dort funktioniert.

Kommunikationsprotokolle sind Formen über welche Computer kommunizieren können. Davon gibt es eine Menge, aber um Webseiten zu schreiben ist am wichtigsten HTTP grob zu kennen. HTTP ist ein Anfrage-Antwort Protokoll[12]. Im Folgenden sind die wichtigsten Teile eines HTTP Requests aufgeführt.

HTTP Request	
Methode und Zielressource	Die **Methode** entspricht bloss einer Deklaration dessen, was der Request verursachen will. Die wichtigsten sind vermutlich GET und POST. GET will etwas, das auf dem Server liegt, erhalten, und POST will etwas dem Server geben und damit serverseitig etwas verändern. Es gibt noch weitere Methoden, wie HEAD, PUT, PATCH oder DELETE. Die **Zielressource** entspricht der Zeichenfolge hinter der Topleveldomain und beschreibt die Route zur Zielressource, bspw.: `/index.html`. Die Zielressource ist meist eine Datei wie `index.html`. Jedoch muss es sich nicht unbedingt um eine Datei handeln. Der Server kann die Ressource auch aus der Route generieren. Was genau eine gewisse angefragte Route zurücksendet, hängt letztlich vom Verhalten des Servers ab. Die Serverlogik für eine bestimmte Route kann auch weitere Elemente des Requests wie die Headers oder den Body in Betracht ziehen. Sie kann frei programmiert werden. Ein Server könnte etwa auch bei einem POST-Request lediglich eine Antwort zurücksenden, ohne etwas intern zu verändern.
Headers	Die Request Headers enthalten Metadaten zum Client und zur Anfrage und zur gewünschten Antwort. Die zwei Header mit denen man vielleicht am häufigsten Kontakt haben wird sind: • **Content-Type**: Gibt das Format des Request Bodys an. (z.B. `text/html`, `application/json`, `multipart/form-data`). • **Accept**: Gibt an, welches Format der Client bereit ist in der Antwort zu empfangen. (z.B. text/html, application/json).
Host	Entspricht dem Server, bzw. dessen URL; `www.beispiel.com`
Body	Entspricht den eigentlichen Daten, die gesendet werden. GET- und HEAD-HTTP-Anfragen enthalten in der Regel keinen Body, und Server ignorieren ihn normalerweise, wenn er vorhanden ist.

[10] "DOM": Document Object Model, durch html bestimmt.
[11] Siehe 'fetch' in Kapitel 4.
[12] Es wird jeweils das Kommunikationspaket der einen Richtung vollständig übertragen, bevor ein Kommunikationspaket der anderen Richtung anfängt. Das ist nicht immer so. Online Chats nutzen beispielsweise Protokolle die eher kontinuierlicher, beidseitiger Kommunikation gleichen, als einem abwechselndes Hin und Her von Anfrage und Antwort.

Und Folgendes wären die wichtigsten Teile einer HTTP Response:

HTTP Response	
Status Codes	HTTP-Statuscodes können durch zusätzliche Ziffern genauer spezifiziert werden. Die folgende Liste zeigt alle möglichen ersten Ziffern von Statuscodes: **1xx:** Information – Die Anfrage wurde empfangen, und die Verarbeitung wird fortgesetzt. **2xx:** Success – Die Anfrage war erfolgreich, und die gewünschte Antwort wird zurückgegeben. **3xx:** Redirection – Die Ressource wurde verschoben, und der Client muss eine neue URL anfragen. **4xx:** Client Error – Ein Fehler in der Anfrage. (z.B. **404:** Nicht gefunden). **5xx:** Server Error – Der Server hat einen Fehler festgestellt, der die korrekte Antwort verhindert (z.B. **500:** Interner Serverfehler).
Headers	Die Response Headers enthalten Metadaten zur Antwort des Servers. Einige häufige HTTP-Response-Header sind: • **Content-Type**: Gibt den Medientyp der Antwort an, z.B. `text/html`, `application/json`, `multipart/form-data`. • **Content-Length**: Gibt die Größe der Antwort in Bytes an. • **Set-Cookie**: Wird verwendet, um Cookies vom Server an den Client zu senden, die für die spätere Kommunikation gespeichert werden können.
Body	Beinhaltet die eigentliche Antwort, die der Server zurückgibt. Dies kann HTML, JSON, Text oder andere Datenformate umfassen, je nach Anfrage und Antworttyp.

Wenn `www.beispiel.com` im Browser eingegeben wird, wird also ein GET-Request gemacht. Im Body der Response wird `index.html`[13] gesendet. Falls das HTML Dokument Referenzen zu weiteren Ressourcen wie CSS-Dateien, JavaScript oder Bildern enthält, fordert der Browser diese Ressourcen separat über zusätzliche GET-Requests an. Sie werden nicht automatisch in einem einzigen Response-Body mitgeschickt. Jede Ressource wird eigenständig angefragt und erhält eine eigene HTTP-Response vom Server.

Wenn der Content-Type application/json gesetzt ist, versucht der Server nicht daraus eine Website zu zeichnen. Denn es handelt sich rein um Daten. Bei **JSON** handelt es sich jeweils um einen sogenannten "Key", oder Bezeichnung der Variable, und dem Variablenwert selbst. JSON kann beliebig geschachtelt werden, und hat bspw. folgende Form:

```
{
  "name": "Beispielfirma",
  "adresse": {
    "strasse": "Hauptstrasse 1",
    "stadt": "Chur",
    "postleitzahl": "7000"
  }
}
```

Routen die solche Antworten generieren, werden oft API[14] genannt. Mit einer Anfrage an einen solchen "Endpunkt" kann beispielsweise das Javascript einer geladenen Seite gewisse Daten einer Website aktualisieren, ohne dass eine ganze Seite neu geladen werden müsste. Bloss Javascript und die API müssen arbeiten.

[13] Server sind typischerweise so konfiguriert, dass wenn eine Anfrage ohne genauere Route ankommt, intern ein File namens "index" gesucht und gesendet wird.

[14] Application Programming Interface. Das bezeichnet irgendeine Form von Software-Schnittstelle, die es einer Software ermöglicht, strukturiert Daten oder Funktionen von einer anderen Software anzufordern und zu empfangen.

2 HTML

HTML deklariert die Struktur einer Website über "tag's". Ein tag ist ein Bestandteil der HTML-Syntax, der in spitzen Klammern eingeschlossen ist. Tags können direkt Text enthalten, oder sie können *beliebig* geschachtelt werden. Ein tag muss meistens geöffnet und geschlossen werden (`<bsp>someText_orOtherTags</bsp>`). Ein typisches HTML Template sieht wie folgt aus;

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Title</title>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

Ganz aussen umspannt die Deklaration dass es sich um html handelt alles. Darin befindet sich der `head` und `body`. Der Anfang inklusive dem `head` entspricht Metadaten wie der genutzen Sprache, Characterset, Titel[15] und möglicherweise viel weiteres. Der `body` entspricht den Inhalten die tatsächlich gerendert werden. Dieser besteht wiederum oft aus `header`, `main` sowie `footer`. Es gibt sehr viele weitere HTML-tags[16] und mögliche zugehörige Attribute[17], daher sind im folgenden nur einige der wichtigsten ausgewählt worden.

1. `<div>`:
   Der `<div>`-Tag ist eigentlich an sich bedeutungslos, bis darauf dass er ein tag ist, und daher zum gruppieren anderer HTML Elemente genutzt werden kann. Beispiel:

   ```
   <div>
       <p>Dies ist ein Absatz innerhalb eines Divs.</p>
   </div>
   ```

2. `<p>`:
   Der `<p>`-Tag definiert einen Textabsatz. Es sorgt standardmäßig für korrekte Textabstände und Ausrichtung. Beispiel:

   ```
   <p>Dies ist ein Textabsatz.</p>
   ```

3. `<form>`:
   Der `<form>`-Tag erstellt ein interaktives Formular zur Erfassung von Benutzereingaben. Es unterstützt eine Vielzahl von Eingabetypen, wie Textfelder, Kontrollkästchen und Schaltflächen. Beispiel:

   ```
   <form action="/submit" method="POST">
       <input type="text" name="username" placeholder="enter
   username">
       <button type="submit">send</button>
   </form>
   ```

[15] Dieser Titel würde nicht auf der Website selbst gerendert, würde aber im Browser, oben im Tab der entsprechenden Website, angezeigt werden.
[16] Siehe chatbots, oder `selfhtml.org` oder `w3schools.com`
[17] Ein Attribut ist im Prinzip ein Parameter des zugehörigen Tags.

4. `<canvas>`:
   Das `<canvas>`-Tag bietet einen Zeichenbereich in HTML zur Darstellung von Pixelgrafiken, häufig verwendet mit JavaScript-Bibliotheken wie Chart.js oder benutzerdefinierten Zeichen-Skripten. Beispiel:

   ```
   <canvas id="myCanvas" width="400" height="400"></canvas>
   ```

5. `<a>`:
   Das `<a>`-Tag definiert einen Hyperlink und löst beim Klick standardmässig einen HTTP-Request an die im `href` Attribut gegebene URL aus. Beispiel:

   ```
   <a href="https://www.example.com">Beispiel besuchen</a>
   ```

6. `<img>`:
   Der `<img>`-tag dient dazu, Bilder in HTML als Teil des Inhalts einzufügen. Es erfordert das Attribut `src`, das die Bildquelle angibt, sowie optional `alt` für eine Beschreibung des Bildes. Beispiel:

   ```
   <img src="path/to/image.jpg" alt="Beschreibung des Bildes">
   ```

Man kann bestimmten Elementen[18] eine `class` oder `id` zuweisen. Dabei ist die Idee sie zu gruppieren. Allerdings nicht so, dass es die Seitenstruktur beinflusst, denn jene Gruppierung ist in Verbindung mit dem CSS bedeutsam. CSS weist grundsätzlich bestimmten Mengen an Elementen bestimmte Darstellungsparameter zu. Um eine solche Menge zu definieren kann man man mehreren tags die selbe `class` zuweisen (`<bsp class="someClass">`). Wenn man nur einen tag ansprechen will, kann man ihm eine `id` zuweisen (`<bsp id="someId">`). Tags haben aber meistens auch schon ohne CSS Darstellungsparameter die ihnen vom Browser zugewiesen werden. Jene können aber im CSS überschrieben werden.

[18] Damit meint man eine bestimmte Verwendung eines tags im Code, nicht den tag als solches.

3 CSS

CSS dreht sich um Regeln. Wie bereits erwähnt bestehen jene aus der Selektion bestimmter HTML-Elemente und der Zuweisung von Darstellungs-properties. CSS könnte man also als *deklarativ* bezeichnen[19], da damit dem Browser deklariert wird wie etwas sein soll. Im Gegensatz dazu könnte man "richtige" Programmiersprachen wie Javascript eher als *instruktiv* bezeichnen, da mit ihnen beschrieben werden kann was getan werden soll, nicht wie etwas sein soll. CSS wird normalerweise[20] in einer separaten Datei gespeichert und dann im head mit dem link-tag[21] verknüpft. Denn so wäre es möglich sie in mehreren HTML-Dokumenten wiederzuverwenden. Folgendes ist eine Auswahl der wichtigsten Wege bestimmte Element-Mengen anzusprechen:

1. Man kann alle Elemente eines bestimmten Tags ansprechen:

```
p {
    color: blue;
    font-size: 16px;
}
```

2. Man kann alle Elemente mit einer bestimmten `class` ansprechen:

```
.exampleClass {
    background-color: red;
    padding: 10px;
}
```

3. Man kann ein Element mit einer bestimmten `id` ansprechen:

```
#exampleId {
    text-align: center;
    font-weight: bold;
}
```

4. Man kann Elemente eines bestimmten Tags und eines bestimmten Attributs ansprechen:

```
input[type="text"] {
    border: 1px solid black;
}
```

5. Man kann alle "child"-Elemente ansprechen. Child-Elemente sind alle Elemente einer bestimmten Eigenschaft (wie eben `tag`, `class` oder `id`), die *in* einem anderen so genanntem "parent"-Element[22] sind. Zuerst wird der parent Selektor geschrieben, und dann der child Selektor. Im folgenden Beispiel wird die Auswahl also zuerst auf alle divs im HTML reduziert, und dann auf alle `p`-Elemente die in einem `div` sind.)

```
div p {
    color: red;
}
```

6. Man kann nur alle *direkten* child-Elemente ansprechen, sprich es werden nur die child-Elemente eines parents angesprochen, aber alle Kinder wiederum jenes childs werden *nicht* angesprochen. Im folgenden Beispiel werden die direkten children des `ul`[23] tags angesprochen.

[19] Auch HTML kann man als deklarativ bezeichnen.
[20] CSS könnte auch in einem style tag im head, oder in style Attributen der zugehörigen Tags ("Inline-CSS") direkt im HTML benutzt werden. Das ist allerdings meist nicht die sauberste Variante CSS einzubinden.
[21] `<link rel="stylesheet" href="relativePath_to_styles.css">`
[22] Oft auch "container" oder "wrapper".
[23] Unordered list.

```
ul > li {
    list-style-type: square;
}
```

Es gibt ebenfalls Pseudo-classes. Jene selektieren nicht statische Elemente des HTML's, sondern temporäre Zustände[24] eines Elements. Daher sind sie "pseudo", denn sie sind eigentlich nicht classes, bzw. "entstehen" erst wenn der Browser den selektierten Zustand erkennt. Sie ermöglichen einen ersten Schritt dahin eine Website dynamisch zu machen.

```
button:hover {
    background-color: blue;
}
```

Und es gibt Pseudo-Elemente. Jene selektieren bestimmte Teile eines HTML-Elements, wie z. B. den ersten Buchstaben oder die erste Zeile eines Textes.

```
p::first-letter {
    font-size: 2em;
}
```

CSS Properties gibt es sehr viele, und sehr viele verschiedene Interaktionen zwischen ihnen, und oft ist unerwartetes Verhalten etwas undurchsichtig. Daher macht es nicht Sinn hier zu versuchen CSS Properties[25] umfassend zu konzeptualisieren, oder eine möglichst grosse Menge aufzulisten. Trotzdem wird im Folgenden auf die grundlegende Darstellung von Block-Elementen und eine Auswahl der wichtigsten Properties eingegangen. HTML Elemente zugehörig zu tags wie `div`, `main` oder `body` sind standardmässig "Block" Elemente in CSS. Elemente zugehörig zu dem `p` tag oder ähnlichen tags sind es nicht, könnten aber auch so konfiguriert werden, und zwar mit `display: block;`. Jene Block Elemente machen die hauptsächliche Struktur der Darstellung aus. Sie werden rechteckig angezeigt, wie der Name schon sagt. Man kann ihre Breite und Höhe, sowie margin, padding und auch ihre Position in der Ebenen-Reihenfolge[26] anpassen. Ebenfalls kann man ihnen ein Hintergrundbild geben, was statt der Einbindung als echtes HTML-Element genutzt werden kann.

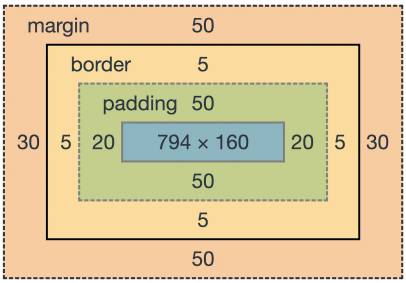

Abbildung 1: CSS Box Modell

Margin, border und padding entsprechen alle Raum um den content eines Blocks herum (blau in Abbildung 1). Welche davon als Angaben für Leerraum ausserhalb der Box und welche als Angaben für Leerraum innerhalb der Box gelten, kann konfiguriert werden. Und zwar kann das mit der Property `box-model` gemacht werden. Das bedeutet, dass die festgelegte Breite und Höhe einer Box, wenn `box-sizing: border-box;` verwendet wird, auch Padding und Border mit einbezieht, während sie bei der Standard-Einstellung `content-box` nur den content betreffen. Das ist eine häufige Quelle von kleinen Ärgernissen, wenn zwei Boxen ähnlich gross sein sollten, es aber nicht ganz sind. Dann kann es sein, dass der unterschiedliche border-box Parameter daran schuld ist. Der border kann eine Dicke oder ein Radius zugewiesen werden, welcher für runde Ecken sorgt. Wenn der Radius auf 50% gesetzt ist, ist der "Block" nun eine Kugel. (Aufführung der Masseinheiten folgt in diesem Kapitel.)

[24] Z.b. "hover", "focus" oder "visited".
[25] Siehe chatbots, selfhtml.org/wiki/CSS oder w3schools.com
[26] Property: "z-index". Mit Ebenen sind hier Ebenen wie in Photoshop gemeint, falls sich zwei Boxen überlappen. Das passiert oft, bspw. nur schon wenn wir ein Element haben was Child ist. (Irgendein `div` im `body` entspräche dem.)

Mit der property `display` kann konfiguriert werden, wie die child Elemente angeordnet werden sollen. Die meisten Gestaltungen lassen sich mit `display: flex;` oder `display: grid;` erledigen. Mit `display: flex;` können, über weitere zugehörige Parameter, die children in einer Linie verteilt werden. Mit `display: grid;` kann ein Raster definiert werden, in welches später die children beliebig eingeordnet werden können. So könnten mit `flex` die children eines ersten Parents horizontal angeordnet werden. Es ist möglich, dass eines jener children selbst children enthält, die beispielsweise entlang der Vertikalen mit flex angeordnet werden können (Siehe Abbildung 2). An diesem Beispiel kann man sich gut vorstellen wie nur mit `flex` schon recht komplexe Layouts gestaltet werden können. Zudem können mit `flex` und `grid` Grössenangaben relativ zur Browserfenstergrösse genutzt werden. Somit kann das Layout dynamisch in die Browserfenstergrösse eingepasst[27] werden. Wenn fixe Masse genutzt würden, wäre die Website immer noch vergleichbar mit einem Dokument, und man müsste in entsprechender Richtung scrollen, in welcher nach Fensteranpassung die Website *nicht* mehr rein passt.

Abbildung 2: `display: flex;`-Demo. Die violetten Bänder verdeutlichen die Verteilung der ersten Ebene an Children.

Wenn man nun mit vorhergehenden Properties ein schön sauber strukturiertes Layout gemacht hat, kann man daraus aber auch wieder ausbrechen. Siehe beispielsweise die `transform` property. Mit ihr kann man, vom gegebenen eigentlichen Ort, das Objekt irgendwie und irgendwohin transformieren.

Es gibt verschiedene **Masseinheiten in CSS**; `px` ist eine absolute Einheit und entspricht einem einzelnen Pixel der Anzeige, skaliert jedoch mit der Browser-Zoomstufe. `rem` ist eine relative Einheit, die auf der Schriftgrösse des Wurzelelements (`<html>`-Tag) basiert. Der Vorteil von `rem` liegt darin, dass der Browser diese Schriftgrösse des Wurzelelements setzt, trotzdem es nicht gerendert wird. Der Browser kennt die Anzeigegrösse des Displays und versucht, die Schriftgrösse des Wurzelelements so zu setzen, dass sie auf verschiedenen Geräten in einer ähnlichen Größe angezeigt wird. `em` ist ebenfalls relativ, bezieht sich jedoch auf die Schriftgrösse des direkt übergeordneten Elements. `%` gibt Grössen in Relation zu einem übergeordneten Element an und erlaubt dynamische Layout-Anpassungen[28]. `vh` und `vw` definieren Grössen relativ zur Höhe bzw. Breite des Browserfensters. `px` wird zunehmend durch `rem` ersetzt, da letzteres eine bessere Anpassung an unterschiedliche Geräte ermöglicht. Für Schriften sollte man also immer `rem` verwenden, und für Box-Grössen ebenfalls `rem` oder eine relative Einheit wie `vh`, `vw` oder `%`.

Es gibt auch spezielle Werte wie `auto`, die es dem Browser ermöglichen, die Größe von Elementen dynamisch zu berechnen. So könnte man z. B. die `margin` einer Box links und rechts "offen" lassen: `margin: 0 auto`. Dann würde der Browser diese Box horizontal in der Mitte des übergeordneten Elements positionieren. Ähnlich kann man die Dimension eines Parents auf den Content anpassen, z. B. mit `height: fit-content`. Der Content muss jedoch eine feste Größe haben, damit eine tatsächliche Größe für das Parent-Element ermittelt werden kann. Falls es widersprüchliche Angaben gibt, wie z. B. unbestimmte Größen, löst der Browser dies in der Regel trotzdem irgendwie sinnvoll auf.

Man sieht also ein wenig, dass CSS in der Anwendung meist am wenigsten klar kontrollierbar ist. Das ist eine ähnliche Aussage wie das CSS "deklarativ" ist. Denn beides beschreibt wie CSS ein wenig dem Browser an den Kopf geworfen wird: Was der aber damit anfängt, in Fällen von Unklarheiten oder gar Entwickler-seitiger

[27] siehe Responsiveness weiter unten.
[28] "responsive."

Fehler, ist aber immer noch Sache von ihm selbst. Deshalb; zum Debuggen von CSS ist die Manipulation und direkt sichtbare Umsetzung der Änderungen durch die Entwicklertools[29] eines Browsers wie Firefox oder Chrome sehr nützlich.

Media Queries entsprechen der Möglichkeit ein Layout responsive zu machen, welche am meisten ermöglicht. Mit ihnen kann man CSS-Regeln nur unter bestimmten Bedingungen anzuwenden. Z.b. bei einer bestimmten Bildschirmabmessung oder Ausrichtung des Geräts. An sich könnte man in einer solchen Regel die Darstellung komplett überschreiben und umbauen. Beispielsweise sorgt: `@media (max-width: 600px) { ... }` dafür, dass bestimmte Styles nur auf Geräten mit einer maximalen Breite von 600px angewendet werden.

Variablen machen CSS code wartungsfreundlicher. Daher sollten sie an den Anfang geschrieben werden, und mit aussagekräftigen Namen versehen werden. So:

```
:root {
  --primary-color: #3498db;
  --padding-size: 20px;
  --font-size-large: 1.5rem;
}
```

Mit `-variablenname: wert;` können Variablen definiert werden. Diese Variablen können dann im gesamten Stylesheet verwendet werden, indem man `var(-variablenname)` anstelle des eigentlichen Werts einsetzt. **calc** erlaubt es, mathematische Berechnungen direkt in CSS zu verwenden, wie z.B. `width: calc(100% - var(-variablenname));`, um Werte zu berechnen, die auf anderen Größen basieren. Dadurch können Layouts noch flexibler und präziser gestaltet werden.

[29] Siehe Kapitel 6.

4 Javascript

Javascript's **Variablen** muss kein Datentyp (int, bool, float, string) zugeordnet werden. Jener wird vom Browser während der Interpretation bestimmt, und es dürfen sogar komplexere Datenstrukturen, wie ein Objekt, einer Variabel zugewiesen werden, welche vorher bspw. ein `int`, sprich einer ganzen Zahl, entsprach. Anstelle des Datentypen wird entweder `let`, `var` oder `const` geschrieben. `Let` deklariert die Variable als änderbar, und sie ist innerhalb des Blocks[30] erreichbar. `Var` (veraltet) deklariert ebenfalls eine Variable als änderbar, allerdings ist sie innerhalb der Funktion in der sie geschrieben wird gültig (weiterer Gültigkeitsbereich als "block-scope"). `Const` deklariert eine Variable als konstant, sprich sie darf später nicht mehr angepasst werden, ansonsten würde der Browser, oder vielleicht bereits der Texteditor, sich beklagen (siehe Kapitel 6). Eine `const` Variable hat ebenfalls block-scope.

Mit einer **Funktion** definiert man bestimmten Code um ihn später wieder zu verwenden. Eine Funktion muss der JS-Engine bekannt sein, bevor er sie beim interpretieren akzeptiert, und sie später ausgeführt wird. Daher müsste die Funktionsdefinition vor ihrer Verwendung stehen, typischerweise am Anfang. Das ist auch der Ordnung halber eine gute Angewohnheit. Allerdings muss man bspw. nicht darauf achten, dass eine Funktionsdefinition, die eine andere Funktion aufruft, auch wirklich nach jener steht. Denn beim interpretieren geht die JS-Engine zuerst durch und schreibt den Text so um, dass alle Funktionsdefinitionen vor ihrem ersten Aufruf stehen.

Funktionen können mit "Argumenten", oder Input-Werten definiert werden. In Javascript kann man auch Funktionen als Argumente übergeben[31]:

```
function runCallback(callback) {
  callback();
}

runCallback(function() {
  console.log("Callback wurde aufgerufen!");
});
```

Die `map`-Funktion ist ein praktischeres Beispiel für das Übergeben einer Funktion. Sie ist eine native[32] Methode[33] für Arrays. Sie nimmt eine Callback-Funktion als Argument, wendet diese auf jedes Element des ursprünglichen Arrays an und gibt einen neuen Array mit den Ergebnissen zurück. Der ursprüngliche Array bleibt unverändert.

```
const numbers = [1, 2, 3, 4];
const squared = numbers.map(num => num * num);
console.log(squared); // [1, 4, 9, 16]
```

[30] Der Block ist der Bereich innerhalb zweier geschweifter Klammern. Bspw. innerhalb eines if's oder einer Funktionsdefinition.
[31] Entspricht vermutlich der Übergabe eines function-pointers in C++.
[32] Bezüglich einer Programmiersprache meint man damit Ausdrücke die der Interpreter oder Compiler von sich aus versteht. Meistens erwähnt man das aber nur bei Dingen, die nicht der Grundfunktionalität einer gängigen Programmiersprache entsprechen, sondern auf jener aufbauen oder irgendwie darüber hinaus gehen.
[33] "Methode" sobald eine Funktion Teil eines Objekts ist. Siehe folgende Ausführungen bezüglich Objekte.

Funktionen können auf viele Arten geschrieben werden[34]. Bspw. ist die "arrow"-Syntax oft anzutreffen, und es ist gut sie als Funktionsdefinition zu erkennen. Hier ist dieselbe Funktion in zwei unterschiedlichen Schreibweisen:
(Mit // macht man in Javascript ein Kommentar bis zum Ende der Zeile.)

```
// Konventionelle Syntax
function add(a, b) {
  return a + b;
}

// Arrow-Syntax
const add = (a, b) => a + b;

// Funktionsaufruf waere fuer beide Schreibweisen gleich:
let c = add(1, 1);
```

Objekte sind Datenstrukturen, also Speicherblöcke, bestehend aus mehreren Variablen. In den meisten Programmiersprachen müssen jene Komponenten erstmal definiert werden, bevor das Objekt instantiiert werden darf. In Javascript dürfen zusätzlich zu den üblichen Datentypen sogar Funktionen[35] Komponenten eines Objekts sein. Die folgenden Ausführungen über Objekte ist hauptsächlich als Vorbereitung für den Abschnitten zu Libraries und Asynchronität gedacht.

In Javascript müssen Variablen ja keinen Datentyp haben[36], daher wird das für Objekte auch nicht benötigt. Die Namen der Komponenten (existieren nur im Code) werden *Keys* genannt, und die entsprechenden Werte *Values*. Es darf geschachtelt werden; Zu einem Key dürfen wiederum Key-Value Paare gehören, nicht bloss ein einziger Value.

Im folgenden Beispiel wird eine zugehörige Funktion definiert. Eine solche, so genannte "Method", wird nicht als Teil der Datenstruktur geschrieben, wie es einem erscheinen könnte. Trotzdem kommt ihnen im Vergleich zu normalen Funktionen ein Sonderstatus zu. Denn der JS-Engine wird so deklariert, dass eine solche Funktion über das `this` keyword Zugriff auf die Datenstruktur, zu der sie im Code geschrieben wurde, hat.

```
class car {
    constructor(b, m, y) {
        this.brand = b;
        this.model = m;
        this.year = y;
        console.log('Car constructor called.');
    }
    start() {
        console.log('Brum, brum, engine of the ' + this.brand + ' ' +
    this.model + ' started.');
    }
}
```

[34] Allerdings ist es für den Anfang, und vielleicht auch *meistens*, besser einfacher lesbaren Code zu schreiben, anstatt sehr kurzen Code mit viel moderner Syntax zu schreiben. Performance-massig gibt es meist sowieso keinen Unterschied.
[35] Entspricht vermutlich tatsächlich nicht der Funktion selbst, sondern auch wieder einem function-pointer.
[36] Ohne Datentyp der Komponenten ist es nicht möglich einen kontinuierlichen Speicherblock für das Objekt zu vergeben, denn dazu müsste man die Gesamtlänge wissen. Zudem kann in Javascript der Datentyp der Felder dynamisch verändert werden. Das bedeutet vermutlich, dass Objekte in Javascript nicht wirklich einem kontinuierlichen Block and Speicheradressen entspricht. Stattdessen wird die JS-Engine wohl irgendwie nicht zusammenhängende Speicherblöcke für die Komponenten verwalten.

Über das keyword `new` kann das Objekt instantiiert werden, entsprechend dem `constructor`: `const meinAuto = new Auto("Toyota", "Corolla", 2010);`. Values (Ebenso Methods) einer bestehenden Objekt Instanz können mithilfe des Keys erreicht werden: (`console.log(meinAuto.brand);` resultiert in einem Konsolen log von 'Toyota'.

Die Zusammensetzung eines Objekts im Code explizit zu definieren ist nützlich, wenn man es später mehrmals instantiieren will. In Javascript muss man das aber nicht zwingend tun. Der folgende Code veranlasst, dass direkt eine entsprechende Datenstruktur geschrieben wird:

```
const car = {
  brand: 'Toyota',
  model: 'Corolla',
  year: 2010,
  start() {
        console.log('Brum, brum, engine of the ' + this.brand + ' ' +
   this.model + ' started.');
  }
};
```

Eine **Library**[37] ist ganz einfach Code. Mit einer Menge von Funktionen und Objekten mit Methoden und so weiter. Jener kann in den eigenen Code eingebunden werden. Meist mit einem weiteren Skript tag im html, in welchem eine URL zum Code angegeben wird, oder ein Pfad auf ein lokales File. Dabei muss darauf geachtet werden, dass Skripts welche eine Library benötigen, nach, nicht vor, jener Library eingebunden werden. Wenn man eine Library mit einer URL einbindet hat man den Vorteil, immer die aktuelle Version der Library zu haben. Man kann es aber auch bevorzugen, seine Website nicht von externen Servern abhängig zu machen.

Libraries benutzen oft grosse Objekte zum zentralisieren von viel Funktionalität. Chart.js zum zeichnen von diversen Diagrammen ist ein gutes Beispiel. Durch die Library ist die Definition mit Konstruktor eines Chart-Objekts eingebunden. Dem Konstruktor wird ein Kontext-Objekt mitgegeben. Dafür wird ein canvas html tag Objekt genutzt, sowie noch verarbeitet. (`.getContext('2d')` Was das tut und wieso es benötigt ist, kann einem eigentlich egal sein. Das kann es oft, beim nutzen von Libraries.

Dem Konstruktor wird ebenfalls ein Objekt mit diversen Konfigurationsdaten mitgegeben. Der Konstruktor des Chart-Objekts muss wohl erst jene Konfigurationsdaten in das neue Objekt schreiben, sowie dann die Chart tatsächlich zeichnen. Beispiel Chart.js:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
    // Zugriff auf das zentrale "Chart"-Objekt
    const ctx = document.getElementById('myChart').getContext('2d');
    const myChart = new Chart(ctx, {
        type: 'bar', // Diagrammtyp
        data: {
            labels: ['Rot', 'Blau', 'Gelb'], // Kategorien
            datasets: [{
                label: 'Beispieldaten',
                data: [12, 19, 3], // Werte
                backgroundColor: ['red', 'blue', 'yellow']
            }]
```

[37] Libraries, Packages, Frameworks sind alle nur ein Haufen von Funktionen und Objekten mit Methoden usw., die auf Grundlage einer fundamentaleren Sprache geschrieben werden. Bibliotheken", ‚Pakete' und ‚Frameworks' sind in erster Linie Aussagen über die Kohärenz bzw. Enge der möglichen Anwendungsfälle. Bibliotheken sind die „lockersten", Frameworks die am meisten anwendungsbereichsspezifischen.

```
        },
        options: {
            responsive: true
        }
    });
</script>
<canvas id="myChart" width="400" height="400"></canvas>
```

Funktionen aus dem einen Skript müssen nicht für andere Skripte im selben HTML Dokument verfügbar sein. Über die `import`, `export` Keywords können nur bestimmte Funktionen für andere Skripte verfügbar gemacht werden. Im folgenden Beispiel werden durch Kommentare die Grenzen der eingebunden Skripte bzw. "Module" angedeutet.

```
// module1.js
export function add(a, b) {
  return a + b;
}

// module2.js
import { add } from './module1.js';
console.log(add(2, 3));  // Ausgabe: 5
```

Auch wenn man selbst grössere Projekte schreibt, lohnt es sich manchmal viel benutzte Funktionen in ein Skript abzutrennen, um sie in mehreren anderen Skripten zu nutzen. Spätestens sobald man dieselbe Funktion in mehreren Skripten schreiben will, sollte man das tun. Der Ordnung halber lohnt sich eine Aufteilung oft schon früher.

Das **DOM** (Document Object Modell) ist eine Datenstruktur, die die HTML-Dokumente als Baum von Objekten repräsentiert, wobei man sich jedes Element (wie `<div>`, `<p>`, etc.) als Knoten im Baum vorstellen kann. Im Browser laufen Prozesse die aus dem DOM kontinuierlich die Darstellung der Website verursachen.

Ein `document` Objekt stellt die Wurzel dieses Baums dar. Die JS-Engine stellt das `document` Objekt zum Zugriff über Javascript zur Verfügung. Über die Methoden von `document` können Objekte instantiiert werden, welche ein bestimmtes HTML-Element repräsentiert, und vom Browser andauernd mit dem DOM synchron gehalten wird:

Mittels `document.querySelector` können Elemente über CSS Selektoren verwendet werden. (Bspw. `const button = document.querySelector('#myButton');`) Mit `.getElementById` und `.getElementsByClassName` kann jeweils mit einer id oder class selektiert werden. Jene Methoden sind schneller als der `.querySelector`.

Sobald man so ein Objekt hat, kann man darüber dessen CSS Eigenschaften beliebig anpassen. Beispielsweise: `button.style.backgroundColor = 'blue';`.

Asynchronität: Im Browser können mehrere Programmprozesse parallel laufen. Nach dem Parsen laufen die meisten Anweisungen eines Skripts im Hauptprozess. Andere laufen auf parallelen Prozessen ("*asynchron*"), zum Beispiel eine "Fetch"-Operation. Damit ist *nicht* echte Parallelität[38] gemeint. Die JS-Engine verwaltet sie nur so dass sie quasi parallel laufen. Ein paralleler Prozess teilt dem Hauptprozess seine Ergebnisse meist über einen so genannten "Promise" mit. Ein Promise in Javascript entspricht circa folgendem Ablauf.

1. Ein (Promise-)Objekt wird geschrieben. Der Konstruktor des Objekts erhält die Funktion als Argument, welche parallel ausgeführt werden soll.

[38] Multithreading.

2. Die Funktion wird parallel ausgeführt, während der Hauptprozess weiter läuft. Wenn die Funktion beendet wird, schreibt sie ihr Ergebnis in das Objekt. (Asynchroner Prozess und Hauptprozess dürfen beide auf das Objekt zugreifen.)

3. Im Hauptprozess läuft nach Ausführung aller "synchronen" Operationen ein so genannter *Event-loop*. Er überprüft andauernd ob in allen Promise-Objekten ein Ergebnis[39] steht. Falls das der Fall ist, führt er die entsprechende Funktion aus. Bei Erfolg führt er die Funktion aus, die unter `.then` im Objekt steht, und bei Misserfolg führt er die Funktion die unter `.catch` im Objekt steht aus. Der Eventlistener übergibt dabei dann das Ergebnis oder den Fehler, welcher ebenfalls ins Promise Objekt geschrieben wurde. So *kann* man die `.then` und `.catch` mit Argumenten schreiben, und jene werden bei Aufruf tatsächlich übergeben.

Folgend ein Code-Beispiel einer solchen asynchronen Operation, bzw. eines "Promise". Es ist rein zur Illustration, und ist kein sinnvolles Beispiel, da er immer rejected wird.

```
const myPromise = new Promise((resolve, reject) => {
  const success = false;

  if (success) {
    resolve("Operation war erfolgreich");
  } else {
    reject("Fehler bei der Operation");
  }
});

myPromise
  .then(result => {
    console.log(result);  // Wird nicht ausgefuehrt, weil Promise
    abgelehnt wird
  })
  .catch(error => {
    console.log(error);  // Wird ausgefuehrt: "Fehler bei der Operation"
  });
```

Fetch ist eine native JavaScript-Funktion, die häufig verwendet wird, um HTTP-Anfragen zu machen. Sie ist asynchron und gibt ein Promise-Objekt zurück, das mit den Daten aus der Antwort oder einem Fehler ausgefüllt wird, wenn die Anfrage abgeschlossen ist. Hier ist ein einfaches Beispiel für fetch:

```
fetch('https://api.example.com/data')
  .then(response => response.json())  // Die Antwort wird in JSON
    umgewandelt
  .then(data => {
    console.log(data);  // Hier wird mit den empfangenen Daten gearbeitet
  })
  .catch(error => {
    console.log('Fehler:', error);  // Fehlerbehandlung
  });
```

Funktionen die in sich Promises, bzw. asynchrone Operationen, haben, müssen mit `async` gekennzeichnet werden. `Async` verursacht dass die Funktion ein "top-level" Promise-Objekt schreibt, und eine asynchrone

[39] Ein Promise kann *pending*, *fullfilled* oder *rejected* sein. Wenn die asynchrone Operation beendet, schreibt sie in das Promise-Objekt *fulfilled* oder *rejected*, wo vorher *pending* stand.

Operation startet die jenes top-level Promise-Objekt als fulfilled oder rejected kennzeichnet, sobald alle "lower-level" Promise-Objekte fullfilled oder rejected sind. So wird festgestellt, wann eine Funktion mit asynchronen Operationen in sich beendet ist. Es kann `await` für asynchrone Operationen benutzt werden. Es verursacht, dass die synchronen Teile der Funktion in der await benutzt wird pausiert werden, bis die asynchrone Operation beendet ist. (Nur jene Funktion wird pausiert, die restlichen dürfen weiter laufen.). Da `await` nur mit asynchronen Operationen benutzt werden darf, darf es auch nur in mit `async` gekennzeichneten Funktionen benutzt werden.

```
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
}

fetchData();
```

Der Browser verwaltet Events wie click, keydown, usw., in eigenen internen Datenstrukturen. Diese Events sind immer mit spezifischen DOM-Elementen verknüpft. Der Event-Loop schaut neben Promise-Objekte ebenfalls andauernd in diese Datenstrukturen und führt die registrierten **Event-Listener** aus, sobald ein Event aufgetreten ist. Um dem Browser zu signalisieren, dass er ein bestimmtes Event auf einem bestimmten DOM Element in sich verwalten soll, gibt es die `.addEventlistener()` Methode. Im folgenden Fall würde einfach jeder Klick auf die gesamte Website eine Ausgabe auslösen.

```
document.addEventListener("click", () => {
    console.log("Event Listener ausgeloest.");
});
```

Localstorage entspricht gespeicherten Daten auf dem Client, zugänglich über Javascript. Die Daten sind von jeder Seite auf derselben Domain lesbar. Jede Webseite hat Zugriff auf den localStorage der eigenen Domain, aber nicht auf den localStorage anderer Domains. Localstorage speichert Daten als Key-Value Paare, wobei beide als String gespeichert werden. Wenn man komplexe Daten wie Objekte oder Arrays speichern will, muss man jene zuerst in einen String, beispielsweise in JSON, umwandeln. Beim auslösen muss es wieder zurück konvertiert werden.

```
// Speichern von Daten im localStorage
let user = { name: "Peter", age: 25 };
localStorage.setItem("user", JSON.stringify(user));

// Abrufen der Daten aus localStorage
let storedUser = JSON.parse(localStorage.getItem("user"));
console.log(storedUser.name); // "Peter"
```

Localstorage ist kein Ersatz für Datenbanken (siehe Kapitel **??**), wird aber trotzdem manchmal um Benutzerpräferenzen über mehrere Sitzungen hinweg zu speichern genutzt.

Errorhandling: Ein try...catch Block wird verwendet, um Fehler die während der Ausführung eines Codes auftreten, zu fangen und zu behandeln: Code im try Block wird ausgeführt. Nach jedem Befehl des Codes werden gewisse Bedingungen zu überprüfen, entweder über manuelles Errorhandling oder durch die JS-Engine (Siehe das übernächste Beispiel und folgende.). Wenn einer detektiert wird, wird bis zum catch block gesprungen, und der Error-String wird in den Speicher geschrieben, und dem catch block "übergeben".

```
try {
  let result = someFunctionThatMightFail();
  console.log(result);
} catch (error) {
  console.log("Fehler aufgetreten: ", error.message);
}
```

Manuelles Errorhandling meint, dass die Bedingung die als Fehler gilt im Code definiert wird:

```
function checkValue(value) {
  if (value < 0) {
    throw new Error("Wert muss groesser oder gleich 0 sein");
  }
  return value;
}

try {
  checkValue(-1);
} catch (error) {
  console.log(error.message);  // Gibt "Wert muss groesser oder gleich 0
    sein" aus.
}
```

Automatisches Errorhandling wird von der JS-Engine selbst übernommen, z.B. bei Fehlern, die durch falsche Syntax oder Laufzeitfehler (z.B. undefined Variablen) entstehen:

```
try {
  let x = undefinedVariable;  // Fehler, weil undefinedVariable nicht
    definiert ist
} catch (error) {
  console.log("Ein Fehler trat auf: ", error.message);
}
```

Errorhandling ist besonders zu beachten, wenn mit externen Ressourcen wie API's oder Benutzereingaben verarbeitet. Daher wird fetch oft mit einem try catch Block verwendet.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');

    if (!response.ok) {
      throw new Error('Netzwerkantwort war nicht ok');
    }

    const data = await response.json();
    console.log(data);

  } catch (error) {
    // Fehler abfangen und behandeln
    console.error('Es gab ein Problem mit der Fetch-Anfrage:', error);
  }
}

fetchData();
```

Hier ist wiederum zu sehen, dass fetch ein Promise Objekt returned; die Variable `response` zeigt im vorhergehenden Beispiel auf jenes Objekt, worin die Eigenschaft `ok` gespeichert ist. Jene wird auf `true` gesetzt, wenn der HTTP-Statuscode[40] der Antwort im Bereich zwischen 200 bis 299 liegt.

[40] Siehe Kapitel 1

5 PHP und Datenbanken

PHP ist eine serverseitige Programmiersprache. Ein PHP Skript wird auf dem Server bei einer HTTP-Anfrage ausgeführt. Über die HTTP-Anfrage können diverse Inputs für das Skript übergeben werden, und mit dem Skript könnte man beliebige HTTP-Antworten verursachen. Beispielsweise HTML Code[41] inkl. Javascript oder CSS, oder bloss Text/JSON. Rein daher kann sich gut vorgestellt werden, dass in einem genügend umfangreichen PHP Skript ganze Websiten stecken können. Beispielsweise ist das bekannte CMS[42] *Wordpress* eigentlich einfach ein riesiges PHP Skript.

Die grundlegenden **Kontrollstrukturen** wie `if`, `else` und Schleifen sind praktisch wie in jeder Programmiersprache. Variablen müssen in PHP immer mit einem `$` beginnen, z. B. `$variable`.

PHP ist eine Skriptsprache, wird also interpretiert. Jene PHP-Umgebung stellt so genannte **Superglobalen** (Variablen) wie `$_GET` und `$_POST` zur Verfügung. Über sie können HTTP-Anfragen der entsprechenden Typen ausgelesen werden. Beispielsweise können darüber Formular User-Inputs in eine Variable und danach geantwortet werden:

```
<?php
//Beispiel fuer die Verarbeitung von GET-Daten
if (isset($_GET['name'])) {
    $name = $_GET['name'];
    echo "Hallo, " . htmlspecialchars($name) . "!";
}

//Beispiel fuer die Verarbeitung von POST-Daten
if ($_SERVER['REQUEST_METHOD'] === 'POST' && isset($_POST['email'])) {
    $email = $_POST['email'];
    echo "Ihre E-Mail-Adresse ist: " . htmlspecialchars($email);
}
?>
```

Hier ist ebenfalls erkennbar, wie man die Antworten zusammenbauen kann. `Echo` signalisiert einen Output. Die **Echo**'s werden bei Ausführung gesammelt, und als eine Antwort geschickt. Ebenfalls wird standardmässig HTML geschickt, es kann aber in der HTTP-Antwort als application/json gekennzeichnet werden:

```
<?php
$data = [
    "status" => "success",
    "message" => "Daten erfolgreich verarbeitet",
    "data" => [1, 2, 3, 4]
];

//Hier wird die HTTP-Antwort entsprechend konfiguriert.
header('Content-Type: application/json');

echo json_encode($data);
?>
```

[41] Es kann ebenfalls andersherum gemacht werden, falls nicht viel PHP code benötigt wird. Ein PHP Skript kann in einer HTML Datei "versteckt" werden. Dazu muss der PHP-Code einfach wie folgt umschlossen werden: `<?php ... ?>`. Zudem muss die Datei nun auf `.php` enden. Bevor jenes HTML bei Anfrage an den Client geschickt wird, wird das PHP ausgeführt, und ermöglicht so eine dynamische Seitengenerierung.

[42] Content-Management System.

Datenbanken sind für Webseiten sehr wichtig. Sie liegen auf dem Server und beinhalten meistens User-Informationen, oder generell wichtige Informationen für Inhalte einer Webseite, die sich über die Zeit ändern können sollen. Eine Datenbank kann man sich meist[43] wie Tabellen ("Tables") vorstellen, bestehend aus row's und column's. Eine row entspricht dabei einer bestimmten Dateneinheit, und in den zugehörigen columns stehen die zugehörige Attribute. Mit den Column's definiert man also die Komponenten der Einheiten, die man in einem bestimmten Table speichern will. Man kann an sich eine Datenbank beliebig durch ein PHP-Skript konfigurieren, aber meistens ist es möglich das einfacher in Websites wie "phpMyAdmin" zu machen, welche vom Webhosting Anbieter angeboten werden:

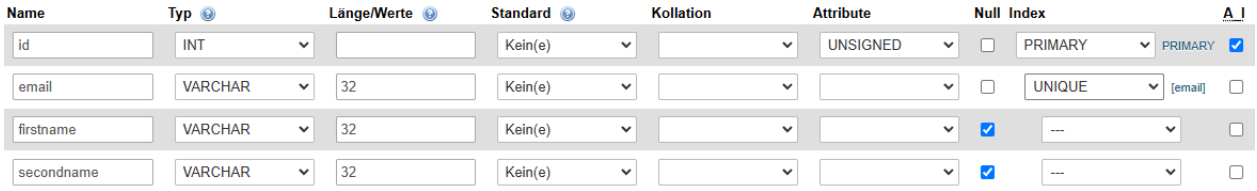

Abbildung 3: Konfiguration eines Tables in phpMyAdmin.

Es würde einen zu grossen Umfang benötigen, hier alle möglichen Column Konfigurations Möglichkeiten durch zu gehen. Hauptsächlich entspricht die Konfiguration dem festlegen des Datentyps, oder ob ein neuer Eintrag jeweils automatisch erzeugt werden soll. Beispielsweise haben die meisten Tables eine Column "ID", zum eindeutigen identifizieren eines Eintrags. Jene kann bei jedem neuen Eintrag automatisch als die fortlaufend nächste Zahl eingetragen werden ("Auto Increment"), und muss nicht von aussen kommen.

Ebenso können "Relations" zwischen Tables definiert werden. Zum Beispiel kann man sich die Situation vorstellen, wo ein Table alle User als Einträge hat, und ein anderer Table alle Posts, die jene User gemacht haben, umfasst. Nehmen wir an, im zweiten Table ist für jeden Post ebenfalls eine "userId" vermerkt, die einer Id in dem User-Table entspricht. Nun kann man eine Relation zwischen jener userId und der Id im User-table machen. Jene könnte so konfiguriert werden, dass der gesamte entsprechende Post gelöscht wird, falls der User mit der entsprechenden Id gelöscht wird. Ebenso könnte die UserId im Table Posts angepasst werden, falls die entsprechende id im User Table verändert würde. Das sollte allerdings in Praxis nie gemacht werden, da die eindeutige Identifizierung eines Eintrag eine sensitive Funktionalität eines Tables ist. Jene Relations können in phpMyAdmin sogar über eine visuelle Oberfläche erstellt werden. ("Designer".)

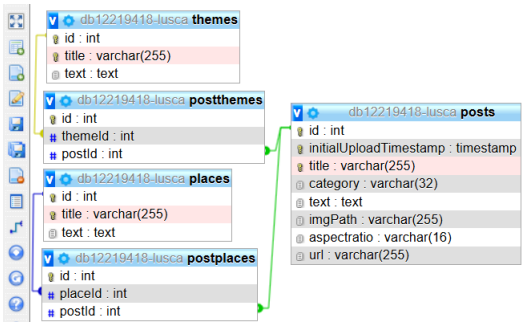

Abbildung 4: Designer Ansicht einer etwas komplexeren Table Struktur, mit Relations als farbige Linien indiziert.

[43] Das entspräche einer so genannt relationalen Datenbank. Non-relationale Datenbanken speichern Daten in dokumentähnlichen Formaten wie JSON. Non relationale Datenbanken erlauben keine Relations zwischen Tables. (Siehe weiter Ausführungen im Abschnitt.)

Datenbank-Server[44] verwalten die Datenbank und all jene Funktionalität. Sie verstehen eine bestimmte Sprache, mit der Auslesen oder Bearbeiten der Daten instruiert werden kann.

In einem PHP Skript kann man mit einem Datenbank-Server über meist SQL[45] kommunizieren. SQL im Detail zu kennen ist für den Anfang nicht nötig und meist schnell durch eine kurze Recherche ersetzbar. Allerdings lohnt es sich doch die häufigen, groben Strukturen zu kennen. Die häufigsten Operationen sind Create, Read, Update und Delete. Ein Beispiel für jede jener Operationen:

```
-- CREATE (‘‘--’’ leitet ein Zeilen-Kommentar ein.)
INSERT INTO users (name, email) VALUES ('peter', 'peter@example.com');

-- READ
SELECT * FROM users;

-- UPDATE
UPDATE users SET email='something_new@example.com' WHERE name='peter';

-- DELETE
DELETE FROM users WHERE name='peter';
```

Der Stern nach `SELECT` meint, dass der ganze Eintrag ausgelesen werden soll. Anstelle des Sterns können bestimmte Spalten-Mengen ausgewählt werden. Mit `WHERE` lassen sich Einträge bzw. rows filtern. Damit lassen sich auch komplexere Filterbedingungen definieren, indem man logische Operatoren und Vergleichsoperatoren, sowie Klammerung für Priorisierung verwendet:

```
SELECT * FROM users
WHERE (age > 18 AND city = 'Chur')
   OR (authorization = 'admin' AND last_login > '2025-01-01');
```

Hier würde müsste nun einer[46] oder beide der Ausdrücke in den Klammern wahr sein, damit ein gegebener Eintrag ausgewählt würde. Die entsprechende Person müsste also 18 Jahre alt sein und in Chur leben, oder sie müsste Admin sein und sich zuletzt am 01.01.2025 eingeloggt haben.

[44] Einem Programm, natürlich, aber meist eines was nur auf dem Computer erreichbar ist, wo auch der Webserver läuft. Und nicht aus dem Netzwerk.
[45] Structured Query Language.
[46] Wenn nur einer wahr sein soll, gibt es das exklusive `OR`; `XOR`.

Die Kommunikation mit dem Datenbank-Server macht man am modernsten über ein PDO[47]. Wie in bei manchen Libraries in javascript, wird hierzu ein Objekt generiert, und darüber die Datenbank Interaktion konfiguriert. Es handelt sich allerdings nicht um eine asynchrone Operation. Es wird ein Statement vorbereitet, welches Platzhalter enthält (z.B. `:name` oder `?`). Die tatsächlichen Werte für jene Platzhalter werden über diesen Weg von dem Datenbank-Server als reine Daten behandelt, und nicht als ausführbarer SQL-Code. Dies verhindert, dass bösartiger Code, der über Eingabefelder eingeschleust wird, ausgeführt werden kann. Die Trennung von SQL-Code und Daten reduziert das Risiko von "SQL-Injection" erheblich.

```
<?php
$dsn = 'mysql:host=localhost;dbname=beispiel_db';
$username = 'root';
$password = 'xy123';

try {
    $pdo = new PDO($dsn, $username, $password);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    $stmt = $pdo->prepare("INSERT INTO users (name, email) VALUES (:name,
    :email)");

    $stmt->execute([':name' => 'Lina', ':email' => 'Lina@example.com']);

    echo "Datensatz erfolgreich eingefuegt.";
} catch (PDOException $e) {
    echo "Fehler: " . $e->getMessage();
}
?>
```

[47] PHP Data Objects.

6 Praktisches

Code Editors wie **VSCode** sind dank vielen Dingen sehr nützlich. Bspw. Code highlighting[48], Code Bereiche zuklappen, sowie den vielen Addons wie SFTP[49], oder für Github[50]. Hier einige der Shortcuts bezüglich Text Editierung, die oft nützlich sind:

Shortcuts und Funktionen	
alt + LMB	Mehrere Cursor setzen, um gleichzeitig an mehreren Stellen zu schreiben.
shift + alt + a	Kommentiere Blöcke aus oder ein.
ctrl + shift + 9	Gehe zum Ende des Blocks. (geschweifte Klammer.)
ctrl + d	addiere nächstes Vorkommnis des selektieren Worts zu Selektion. (geschweifte Klammer.)
tippe: !	HTML template.

In der persönlichen Beobachtung oder Erfahrung des Autors ist es nicht sehr sinnvoll **KI-Chats** umfänglich zu nutzen, da je nach Nutzungsweise die Effektivität tatsächlich sehr niedrig und der Lerneffekt ebenfalls relativ niedrig ist. Speziell um grössere Strukturen zu schreiben, wird "KI"-Code den man selbst nicht sehr genau geschrieben hat, schnell sehr chaotisch und unübersichtlich, und schlecht wartbar.

Stattdessen scheint aber empfehlenswert bestimmte Themen und fragen mit der KI "auszudiskutieren", bis man es verstanden hat. Bestimmte Syntax nachzuschauen, oder Teile schreiben zu lassen, die man funktionell gut versteht, nur syntax-technisch nicht, ist ebenfalls ok.

Eine **regelmässige Formatierung**, und wo nötig auch Kommentierung, des Codes, auch wenn sie nicht für die Code Funktion relevant ist, hilft der Wartbarkeit und der Freude am Code schreiben erfahrungsgemäss stark.

Die **Entwicklertools** sind in Chrome sowie Firefox mit F12 aufzurufen.

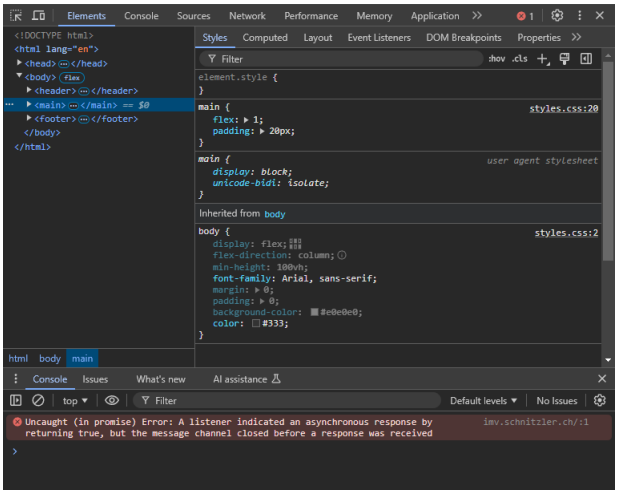

Abbildung 5: Chrome Entwicklertools.

- Mit ihnen kann die HTML Struktur textlich (*Elements*), sowie per Hovern über die Anzeige (Pfeil diagonal in gestrichelten Viereck oben links), nachvollzogen sowie manipuliert werden.
- Ebenso können CSS Regeln addiert oder manipuliert werden (*Styles*), mit direktem Feedback der Effekte.

[48] Meint dass der Code regelmässig einfärbt ist, das hilft bei der Übersicht
[49] lokalen Code über FTP synchron mit Webserver halten.
[50] Eine Art Cloud für Code inkl. Versionierung und ausgeklügelter Zusammenarbeit.

- Die bereits häufig in Beispielen erwähnten Ausgaben von `console.log()` sind unter *Console* einzusehen, sowie auch error Meldungen beider Arten[51]. Ebenfalls kann man in der Konsole Funktionen aus dem Javascript ausführen.
- Der gesamte Sourcecode ist unter *Sources* einzusehen.

Das sind die erfahrungsgemäss am häufig benutzen Funktionen der Browser Entwicklertools. Manchmal sind die Ladezeiten (*Network*) und Renderzeiten (*Performance*) noch interessant.

Um eine Website auf einen neuen Hosting Anbieter umzuziehen ist der Source code offensichtlich nicht das Problem. Aber die Inhalte der Datenbank sind auch entscheidend. Glücklicherweise bietet phpMyAdmin eine Möglichkeit SQL Befehle zu extrahieren, durch welche bestimmte Tables inklusive Inhalt generiert werden können:

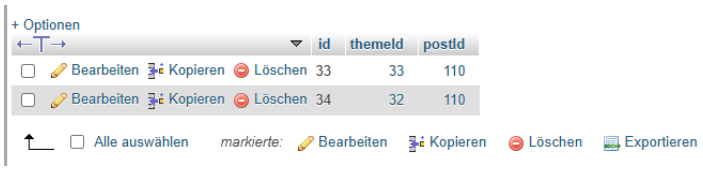

Abbildung 6: Export table to sql.

[51] Siehe Kapitel 4