Tag 5 CSS: Grid

CSS Grid ist eines der leistungsfähigsten Layoutsysteme in CSS. Es erlaubt die Erstellung von **zweidimensionalen Layouts** – also Spalten und Zeilen gleichzeitig – und ist besonders nützlich für komplexe, strukturierte Layouts, wie z. B. Dashboard-Layouts, Seitenlayouts mit mehreren Bereichen oder responsive Raster.

---

1. Einführung in CSS Grid

CSS Grid Layout wurde eingeführt, um eine native, deklarative Möglichkeit zu schaffen, Elemente **zweidimensional anzuordnen**. Es bietet:

- vollständige Kontrolle über Zeilen und Spalten
- Flexibilität durch dynamische Größenvergabe (`fr`, `auto`, `minmax()`)
- klar definierbare Bereiche durch `grid-template-areas`
- einfache Integration mit Media Queries

Unterschied zu Flexbox:

- **Flexbox** eignet sich besser für lineare Layouts (1D)
- **Grid** eignet sich besser für strukturierte Raster (2D)

---

2. Grundstruktur: Grid-Container und Grid-Items

a) Der Grid-Container

Ein Container wird durch `display: grid` (oder `inline-grid`) zum **Grid-Container**:

```
.container {
  display: grid;
}
```

b) Die Grid-Items

Alle direkten Kindelemente eines Grid-Containers sind automatisch **Grid-Items**. Jedes dieser Elemente kann einzeln gesteuert werden – Position, Größe, Ausrichtung etc.

> Nur direkte Kinder eines Grid-Containers sind Grid-Items!

---

3. Zeilen und Spalten definieren

Um mit CSS Grid ein Raster zu erzeugen, müssen **Spalten (columns)** und **Zeilen (rows)** definiert werden. Dafür verwendet man die Eigenschaften `grid-template-columns` und `grid-template-rows` im Grid-Container.

---

a) Spalten definieren mit `grid-template-columns`

Diese Eigenschaft legt fest:

- **Wie viele Spalten** es im Grid gibt
- **Wie breit** jede Spalte sein soll

```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;
}
```

In diesem Beispiel:

- Es gibt **drei Spalten**
- Die mittlere Spalte ist **doppelt so breit** wie die äußeren

Mögliche Einheiten für Spaltenbreiten:

- `px`: feste Breite (z. B. `200px`)
- `%`: relativ zur Breite des Containers (z. B. `50%`)
- `fr`: **Fractional Unit** – verteilt verbleibenden Platz nach Anteilen
- `auto`: passt sich dem Inhalt an
- `minmax()`: definiert minimalen und maximalen Bereich für Spalten

Beispiel mit `minmax()`

```
grid-template-columns: minmax(150px, 1fr) 2fr;
```

> Ermöglicht responsives Verhalten bei gleichzeitiger Begrenzung der Mindestgröße

Die Funktion `repeat()`

Wenn mehrere Spalten gleich aufgebaut sind, kann man sie mit `repeat()` kürzer schreiben:

```
.container {
  grid-template-columns: repeat(3, 1fr); /* drei gleich breite Spalten */
}
```

Man kann auch `repeat()` mit anderen Einheiten kombinieren:

```
.container {
  grid-template-columns: 200px repeat(2, 1fr) 100px;
}
```

Beispiel mit `minmax()` und `auto-fit`

```
.container {
  grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
}
```

> Dieses Muster passt sich flexibel der Containerbreite an und erzeugt responsive Spalten.

---

b) Zeilen definieren mit `grid-template-rows`

Mit `grid-template-rows` wird die **Höhe der Zeilen** festgelegt – analog zu den Spalten.

```
.container {
  grid-template-rows: 100px auto 50px;
}
```

In diesem Beispiel:

- Die erste Zeile ist **fix 100px hoch**
- Die zweite Zeile passt sich **dynamisch** dem Inhalt an

- Die dritte Zeile ist wieder **fix auf 50px** begrenzt

Auch hier gilt:

- `fr`, `auto`, `minmax()`, `repeat()` sind ebenso gültig wie bei Spalten

```
.container {
  grid-template-rows: repeat(3, 150px);
}
```

> Zeilenhöhen beeinflussen nur die direkte Fläche der Grid-Zellen, nicht deren Inhalt – z. B. kann ein hoher Inhalt `auto`-Zeilenhöhe erzeugen

Dynamische Zeilenhöhe mit `min-content` / `max-content`

```
.container {
  grid-template-rows: min-content auto max-content;
}
```

- `min-content`: kleinster möglicher Inhalt (z. B. Textumbruch)
- `max-content`: größte natürliche Breite/Höhe des Inhalts

> **Tipp:** Kombiniere `repeat()` und `minmax()` für flexible responsive Layouts:

```
grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
```

---

4. Abstand: `gap`

Die Eigenschaft `gap` definiert den **Abstand zwischen Spalten und Zeilen** (vormals `grid-gap`).

```
.container {
  gap: 20px; /* gleichmäßig */
  row-gap: 10px;
  column-gap: 30px;
}
```

> Vorteil gegenüber `margin`: Kein „doppelter Abstand“ zwischen benachbarten Grid-Items

---

5. Positionierung im Grid

Mit CSS Grid kannst du einzelne Items präzise innerhalb des Rasters positionieren. Jedes Grid-Item kann genau an bestimmten **Spalten- und Zeilenlinien** ausgerichtet werden. Dabei orientierst du dich nicht an Zellen, sondern an **Grid-Linien** – das ist ein entscheidender Unterschied.

Visualisierung eines Grid-Rasters (3 Spalten × 4 Zeilen)

```
+-----+-----+-----+
|     |     |     |  ← Zeile 1
+-----+-----+-----+
|█████|█████|     |  ← Zeile 2
+-----+-----+-----+
|█████|█████|     |  ← Zeile 3
+-----+-----+-----+
|     |     |     |  ← Zeile 4
+-----+-----+-----+
  ↑     ↑     ↑
  1     2     3   ← Spaltenlinien
```

> In diesem Beispiel erstreckt sich ein Element von **Spalte 1 bis 3** (2 Spalten breit) und **Zeile 2 bis 4** (2 Zeilen hoch).

---

a) Spalten-Positionierung mit `grid-column`

```css
.item {
  grid-column-start: 1;
  grid-column-end: 3; /* endet an Linie 3 → deckt Spalten 1 + 2 ab */
}
```

Kurzschreibweise:

```css
.item {
  grid-column: 1 / 3;
}
```

Tipp: Du kannst auch das Keyword `span` verwenden:

```css
.item {
  grid-column: 1 / span 2; /* starte bei Linie 1, spanne 2 Spalten */
}
```

---

b) Zeilen-Positionierung mit `grid-row`

```css
.item {
  grid-row-start: 2;
  grid-row-end: 4; /* deckt Zeile 2 + 3 ab */
}
```

Kurzschreibweise:

```css
.item {
  grid-row: 2 / 4;
}
```

Auch hier kannst du `span` nutzen:

```css
.item {
  grid-row: 2 / span 2; /* startet in Zeile 2, erstreckt sich über 2 Zeilen */
}
```

> **Hinweis:** Grid-Positionierung basiert immer auf **Linien**, nicht auf Zellen. Du zählst also von **Linie zu Linie** – der Bereich dazwischen ist die sichtbare Fläche.

✨ Weitere Beispiele

Beispiel 1: Item, das nur 1 Zelle (1x1) einnimmt

```css
.item {
  grid-column: 2 / 3;
  grid-row: 3 / 4;
}
```

Beispiel 2: Item, das sich über alle drei Spalten erstreckt

```css
.item {
  grid-column: 1 / 4;
}
```

Beispiel 3: Dynamische Position mit `auto`

```css
.item {
  grid-column: auto / span 2; /* beginne automatisch und nehme 2 Spalten ein */
}
```

> Praktisch, wenn Items dynamisch einsortiert werden sollen, ohne feste Startlinie

---

6. Layout mit `grid-template-areas`

Mit `grid-template-areas` kannst du das Layout einer Webseite **visuell lesbar, semantisch benannt und intuitiv gestaltet** definieren. Es ist besonders nützlich, wenn du eine klare Struktur wie **Header – Navigation – Content – Footer** darstellen möchtest.

Was ist `grid-template-areas`?

Statt mit numerischen Linien (`grid-column: 1 / 3`) zu arbeiten, gibst du deinem Grid eine **zeichnerische Struktur**, in der jeder Bereich einen Namen bekommt. Das erleichtert nicht nur die Planung, sondern verbessert auch die Wartbarkeit deines Codes.

Beispiel: Layout mit 3 Spalten und 3 Zeilen

```
```

```css
.container {
  display: grid;
  grid-template-columns: 200px 1fr 1fr;
  grid-template-rows: 80px 1fr 60px;
  grid-template-areas:
    "header  header  header"
    "nav     content content"
    "nav     footer  footer";
  gap: 10px;
}
```

Was passiert hier?

- **Zeile 1:** Ein durchgehender Header über alle drei Spalten
- **Zeile 2:** Navigation links, Content rechts
- **Zeile 3:** Navigation bleibt links, Footer rechts unten

Diese Struktur liest sich wie ein **Bauplan**, direkt im CSS.

Zuweisung der Bereiche mit `grid-area`

Jedes Grid-Item im HTML bekommt ein passendes Label, z. B.:

```css
.header  { grid-area: header; }
.nav     { grid-area: nav; }
.content { grid-area: content; }
.footer  { grid-area: footer; }
```

Dein HTML dazu könnte so aussehen:

```html
<div class="container">
  <div class="header">Header</div>
  <div class="nav">Navigation</div>
  <div class="content">Content</div>
  <div class="footer">Footer</div>
</div>
```

Regeln bei `grid-template-areas`

- **Alle Namen müssen zusammenhängende Rechtecke bilden.** Du darfst keine L-förmigen oder gesplitteten Bereiche mit demselben Namen definieren.
- **Nicht belegte Zellen** werden mit einem Punkt `.` dargestellt:

```css
grid-template-areas:
  "header header header"
  "nav    content ."
  "footer footer footer";
```

- **Anzahl der Spalten und Zeilen** in `grid-template-areas` muss mit `grid-template-columns` und `-rows` übereinstimmen!

Dynamische Anpassung mit Media Queries

Du kannst `grid-template-areas` auch in Media Queries neu definieren, z. B.:

```css
@media (max-width: 768px) {
  .container {
    grid-template-areas:
      "header"
      "nav"
      "content"
      "footer";
    grid-template-columns: 1fr;
    grid-template-rows: auto;
  }
}
```

> Ergebnis: Mobilgerät zeigt die vier Bereiche untereinander, perfekt fürs Responsive Design

---

7. Ausrichtung im Grid

In CSS Grid kannst du sowohl die **Ausrichtung einzelner Items** innerhalb ihrer Zelle steuern als auch die **Gesamtausrichtung** des Grids im Container beeinflussen. Dadurch entsteht ein hohes Maß an Kontrolle über Layout und Gestaltung – besonders bei responsiven Designs.

Ausrichtung einzelner Grid-Items

Diese beiden Eigenschaften wirken **auf jedes Item einzeln** innerhalb seiner Zelle:

a) `justify-self` (horizontal innerhalb der Zelle)

- Steuert die horizontale Ausrichtung des Inhalts **innerhalb der eigenen Spalte**
- Gültige Werte:
  - `start`: linksbündig (bzw. am Anfang der Spalte)
  - `end`: rechtsbündig (am Ende der Spalte)
  - `center`: zentriert in der Spalte
  - `stretch` (Standard): dehnt sich über die volle Spaltenbreite

```css
.item {
  justify-self: center;
}
```

b) `align-self` (vertikal innerhalb der Zelle)

- Steuert die vertikale Ausrichtung des Inhalts **innerhalb der Zeile**
- Gültige Werte:
  - `start`: oben innerhalb der Zelle
  - `end`: unten innerhalb der Zelle
  - `center`: vertikal zentriert
  - `stretch` (Standard): dehnt sich über volle Zellenhöhe

```css
.item {
  align-self: end;
}
```

> Nützlich, um einzelne Grid-Elemente besonders hervorzuheben oder abweichend zu platzieren

---

Standardausrichtung für alle Items im Grid

Diese Eigenschaften definieren das **Standardverhalten für alle Grid-Items**, sofern `justify-self` / `align-self` nicht überschrieben wird.

a) `justify-items` (horizontal innerhalb der Zellen)

```
.container {
  justify-items: center; /* alle Items horizontal zentriert */
}
```

b) `align-items` (vertikal innerhalb der Zellen)

```
.container {
  align-items: stretch; /* alle Items füllen die Zellenhöhe */
}
```

> Wird häufig zusammen verwendet, um gleichmäßige Ausrichtung über alle Items hinweg zu gewährleisten

Gültige Werte (wie bei `self`):

- `start`, `end`, `center`, `stretch`

Ausrichtung des gesamten Grid-Inhalts im Container

Diese Eigenschaften betreffen **das gesamte Grid** im Container – also wie die **Grid-Zeilen und -Spalten** innerhalb des Containers positioniert werden, wenn **mehr Platz verfügbar ist als nötig**.

a) `justify-content` (horizontal)

- Richtet das gesamte Grid horizontal im übergeordneten Container aus
- Funktioniert nur, wenn der Container **mehr Platz hat**, als das Grid benötigt

Mögliche Werte:

- `start`, `end`, `center`
- `space-between`, `space-around`, `space-evenly`
- `stretch` (nicht in allen Browsern unterstützt)

```
.container {
  justify-content: space-between;
}
```

b) `align-content` (vertikal)

- Richtet das Grid vertikal im Container aus (z. B. wenn weniger Zeilen da sind als Containerhöhe)

```
.container {
  align-content: center;
}
```

> Beide Eigenschaften funktionieren nur, wenn das Grid kleiner ist als sein Elternelement – ähnlich wie `flexbox`-Verhalten bei `justify-content`

🧪 Beispiel – zentriertes Grid mit zentrierten Items:

```css
.container {
  display: grid;
  grid-template-columns: repeat(3, 150px);
  grid-template-rows: repeat(2, 100px);
  justify-content: center;
  align-content: center;
  justify-items: center;
  align-items: center;
  gap: 20px;
  height: 100vh;
}
```

Tipp:

Du kannst innerhalb eines Grid-Items auch **Flexbox** verwenden, um **mehrdimensionale Ausrichtung** zu erreichen:

```css
.item {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

> Kombiniere Grid für das große Layout, Flexbox für die Inhalte innerhalb der Boxen – Best of Both Worlds.

---

8. Responsive Design mit CSS Grid

Flexibles Raster mit Media Queries

```css
.container {
  grid-template-columns: 1fr;
}

@media (min-width: 600px) {
  .container {
    grid-template-columns: repeat(3, 1fr);
  }
}
```

Automatisches Layout mit `auto-fit` / `auto-fill`

```css
grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
```

- `auto-fill`: füllt so viele Spalten wie möglich, auch leere
- `auto-fit`: passt sich aktiven Items an

> Sehr nützlich für Galerien oder dynamische Inhalte

---

9. Übungsaufgaben (WDA-konform, kompetenzorientiert)

Aufgabe 1

Erstelle ein Layout mit drei Spalten, bei dem die mittlere Spalte doppelt so viel Platz einnimmt wie die äußeren.

Aufgabe 2

Setze ein Raster mit unterschiedlichen Zeilenhöhen und nutze flexible Maßeinheiten, um die Inhalte dynamisch darzustellen.

Aufgabe 3

Baue ein semantisches Grid mit benannten Bereichen für Header, Navigation, Content und Footer.

Aufgabe 4

Gestalte ein Layout, in dem ein einzelnes Element zwei Zeilen und zwei Spalten überspannt – positioniere es präzise.

Aufgabe 5

Erstelle ein responsives Grid, das sich bei kleinen Bildschirmen in eine Spalte umwandelt, bei größeren in drei.

Aufgabe 6

Entwirf ein flexibles Galerie-Layout mit automatisch anpassbarer Spaltenanzahl, das sich an die Breite des Containers anpasst.