LU02: Fonts, CSS-Variablen, Cascade & Media Queries

Ziel von LU02: Sie bringen Ihr Projekt „Landingpage Alarado“ optisch näher ans Figma-Design, indem Sie

1. die Schrift (Google Font) einbinden,
2. Designwerte als CSS-Variablen definieren und
3. Mit Media Queries das Responsive-Layout für Mobile umsetzen.

Lernziele

- Sie können eine Google Font per `<link>` einbinden und sinnvoll als `font-family` verwenden.
- Sie verstehen den Unterschied zwischen `px`, `em` und `rem` und setzen Schriftgrössen konsistent um.
- Sie können CSS-Variablen in `:root` definieren und mit `var(…)` wiederverwenden.
- Sie verstehen die CSS-**Cascade** (Reihenfolge/Überschreiben/Vererbung).
- Sie können Media Queries für Responsive Design nutzen und mit DevTools testen.

Fonts im Web

Google Fonts via CDN

In vielen Projekten werden Fonts schnell über ein CDN eingebunden (z.B. Google Fonts) – das ist praktisch für Prototypen und Schulprojekte.

Typischer Aufbau im <head>:

```
<link rel="preconnect" href="https://fonts.googleapis.com" />
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
<link
href="https://fonts.googleapis.com/css2?family=Poppins:wght@400;500;600;700&
display=swap" rel="stylesheet" />
```

Dann im CSS verwenden:

```
body {
  font-family: "Poppins", sans-serif;
}
```

Hinweis zur Praxis: In professionellen Projekten werden Fonts häufig **lokal** im

Projekt gespeichert (Datenschutz/Performance/Offline-Fähigkeit). Für dieses Modul genügt zu Beginn das Einbinden via `<link>`.

Besser in der Praxis: Fonts lokal speichern (kurz)

Mit `@font-face` können Sie eine Schrift von Ihrer eigenen Projekt-URL laden und einen Namen vergeben.

```
@font-face {
  font-family: "Poppins";
  src: url("./fonts/Poppins-Regular.woff2") format("woff2");
  font-weight: 400;
  font-style: normal;
}
```

Wichtige Font-Eigenschaften (Repetition)

- `font-family`: Schrift-Familie (immer mit Fallbacks arbeiten, z.B. `sans-serif`)
- `font-weight`: Schriftschnitt – „Dicke“ (z.B. 400 normal, 700 bold – nicht jede Schrift hat alle Stufen)
- `font-style`: normal/italic
- `text-transform`: Darstellung (uppercase/lowercase/capitalize)
- `line-height`: Zeilenhöhe (Best Practice: **ohne Einheit**, z.B. 1.4)

Einheiten bei Fonts: px vs em vs rem (mit Accessibility)

Kurzdefinitionen

- **px**: „fixe“ Design-Einheit. 16px bleibt rechnerisch 16px.
- **em**: relativ zur **berechneten Schriftgrösse** (`font-size`) des **aktuellen Elements**.[1)]
- **rem**: relativ zur **Root-Schriftgrösse** (`font-size` von `html`).

Warum em/rem oft besser sind als px

1) Skalierung & Nutzer-Einstellungen (Accessibility)

Viele Nutzerinnen und Nutzer verändern bewusst die **Standard-Schriftgrösse** im Browser oder verwenden **Zoom** bzw. Betriebssystem-Einstellungen (z.B. Sehhilfe, grosse Schrift).

- Mit **rem/em** „gehen“ Ihre Schriftgrössen **automatisch** mit dieser Einstellung mit, weil sie relativ gerechnet werden.
- Mit **px** bleiben Sie oft näher an einem „fixen“ Design – das kann dazu führen, dass Text im

Verhältnis zu Layout-Abständen schlechter skaliert.

2) Konsistenz über das ganze Projekt (Designsystem)

Mit **rem** können Sie ein klares System bauen:

- Basis ist `html { font-size: … }`
- Alles andere wird in rem definiert (z.B. h1 = 4rem, p = 1.125rem)

Damit können Sie das „Gesamtbild“ sehr kontrolliert skalieren, ohne jedes Element einzeln anzufassen.

3) Robustheit bei Komponenten

em ist besonders nützlich innerhalb von Komponenten:

- Buttons, Badges, Cards können Abstände (padding/margins) in `em` bekommen,
- so bleiben sie proportional zur Textgrösse der Komponente.

Beispiel: Wenn ein Button-Text grösser wird, wächst das Padding automatisch mit – das wirkt typografisch sauber und verhindert „gequetschte“ Buttons.

px – wann ist es trotzdem okay?

px ist nicht „verboten“, aber Sie sollten wissen, wofür:

- **sehr feine Linien** (z.B. 1px border)
- **pixelgenaue Details**, wenn es wirklich fix sein muss
- einzelne Sonderfälle im Layout

Viele moderne Designsysteme nutzen rem für Typografie + Spacing und px nur für kleine, technische Details.

em vs rem – wann welches?

- **rem**: für **globale Typografie** (h1, p, nav, Standardtext) → konsistent im ganzen Projekt
- **em**: für **komponenten-interne Proportionen** (Padding, Icon-Grösse, kleine Abstände) → skaliert mit der Komponente

line-height

Für Lesbarkeit ist eine passende Zeilenhöhe zentral.

- Best Practice: **einheitenlos**, z.B. `line-height: 1.4;`

Einheitenlose line-height skaliert automatisch mit der Schriftgrösse und ist darum sehr stabil –

besonders bei Responsive und bei Nutzer-Zoom.

CSS-Variablen (Custom Properties)

Was sind CSS-Variablen?

CSS-Variablen (korrekt: Custom Properties) sind benannte Werte, die Sie **einmal zentral** definieren und dann überall mit `var(…)` verwenden. Das spart Copy/Paste und macht Anpassungen viel einfacher.

Beispiel:

```
:root {
  --primary: #263fa9;
  --text: #223344;
}
body {
  color: var(--text);
}
button {
  background: var(--primary);
}
```

Warum ist das nützlich?

- **Wiederverwendbarkeit**: Farben/Grössen müssen nicht überall kopiert werden.
- **Wartbarkeit**: Eine Änderung in `:root` wirkt im ganzen Projekt.
- **Semantik**: –`main-text-color` ist verständlicher als #223344.

> **Vergleich zu Print-Design (z.B. InDesign):**
> CSS-Variablen sind ähnlich wie **Absatz-/Zeichenstile** oder ein **Designsystem**: Sie definieren Regeln einmal zentral und wenden sie überall an.

CSS Cascade (Kaskade) – verständlich & wichtig

> **Was ist die Kaskade?**
> Die CSS-Kaskade ist das Regelwerk, das entscheidet, **welche CSS-Regel am Ende wirklich angewendet wird**, wenn mehrere Regeln dasselbe Element betreffen.
> Kurz: Wenn CSS „streitet“, sagt die Kaskade, **wer gewinnt**.

Warum ist das wichtig?

In echten Projekten schreiben Sie nicht nur 1 Regel pro Element:

- Sie haben Basis-Styles (z.B. `body`)
- Komponenten-Styles (z.B. `.menu a`)
- Zustände (z.B. `.active`, `:hover`)
- Responsive Regeln (Media Queries)
- und manchmal überschreibt man später etwas gezielt

Ohne Kaskade wäre unklar, was am Ende gilt.

1) Vererbung (Inheritance): Was „erbt“ ein Element?

Einige CSS-Eigenschaften werden vom Eltern-Element an Kinder **weitergegeben**, wenn das Kind selbst nichts anderes definiert.

Typisch vererbt (sehr häufig):

- `color`
- `font-family`, `font-size`, `font-weight`, `font-style`
- `line-height`
- `text-transform`

Typisch NICHT vererbt:

- `background-color` / `background-image`
- `margin` / `padding`
- `border`
- `width` / `height`
- `display`

Beispiel:

```css
body {
  font-family: "Poppins", sans-serif;
  color: #223344;
  line-height: 1.4;
}
```

Alle Textelemente im Body erhalten diese Werte, solange sie nicht selbst etwas anderes setzen.

2) Überschreiben (Overriding): Wer gewinnt bei Konflikten?

Wenn mehrere Regeln dieselbe Eigenschaft setzen (z.B. `color`), muss CSS entscheiden, welche gilt. Das passiert in dieser Reihenfolge:

Regel 1: Wichtigkeit (''!important'')

- `!important` gewinnt fast immer – wird aber nur in Ausnahmefällen empfohlen.

[2)]

Regel 2: Spezifität (Specificity)

Je „genauer“ ein Selektor ist, desto stärker ist er.

Grob von schwach nach stark:

1. Elementselektor: `p` , `a` , `header` (schwach)
2. Klassenselektor: `.menu`, `.active` (stärker)
3. ID-Selektor: `#hero-section` (sehr stark)

Beispiel:

```
a { color: black; }          /* schwach */
.menu a { color: gray; }     /* stärker */
a.active { color: blue; }    /* ähnlich stark wie .menu a (Klasse + Element)
*/
#special a { color: red; }   /* sehr stark */
```

Regel 3: Reihenfolge im CSS (Last Rule Wins)

Wenn zwei Regeln **gleich spezifisch** sind, gewinnt die Regel, die **später im CSS** steht.

Beispiel:

```
.menu a { color: gray; }
.menu a { color: green; } /* gewinnt, weil später */
```

Media Queries und Kaskade: „CSS gilt nur wenn Bedingung stimmt“

Media Queries sind keine „eigene Welt“ – sie folgen denselben Regeln wie oben. Sie sind wie zusätzliche Regeln, die nur aktiv werden, wenn die Bedingung erfüllt ist.

Beispiel:

```
#hero-section { flex-direction: row; }

@media screen and (width < 1024px) {
  #hero-section { flex-direction: column-reverse; }
}
```

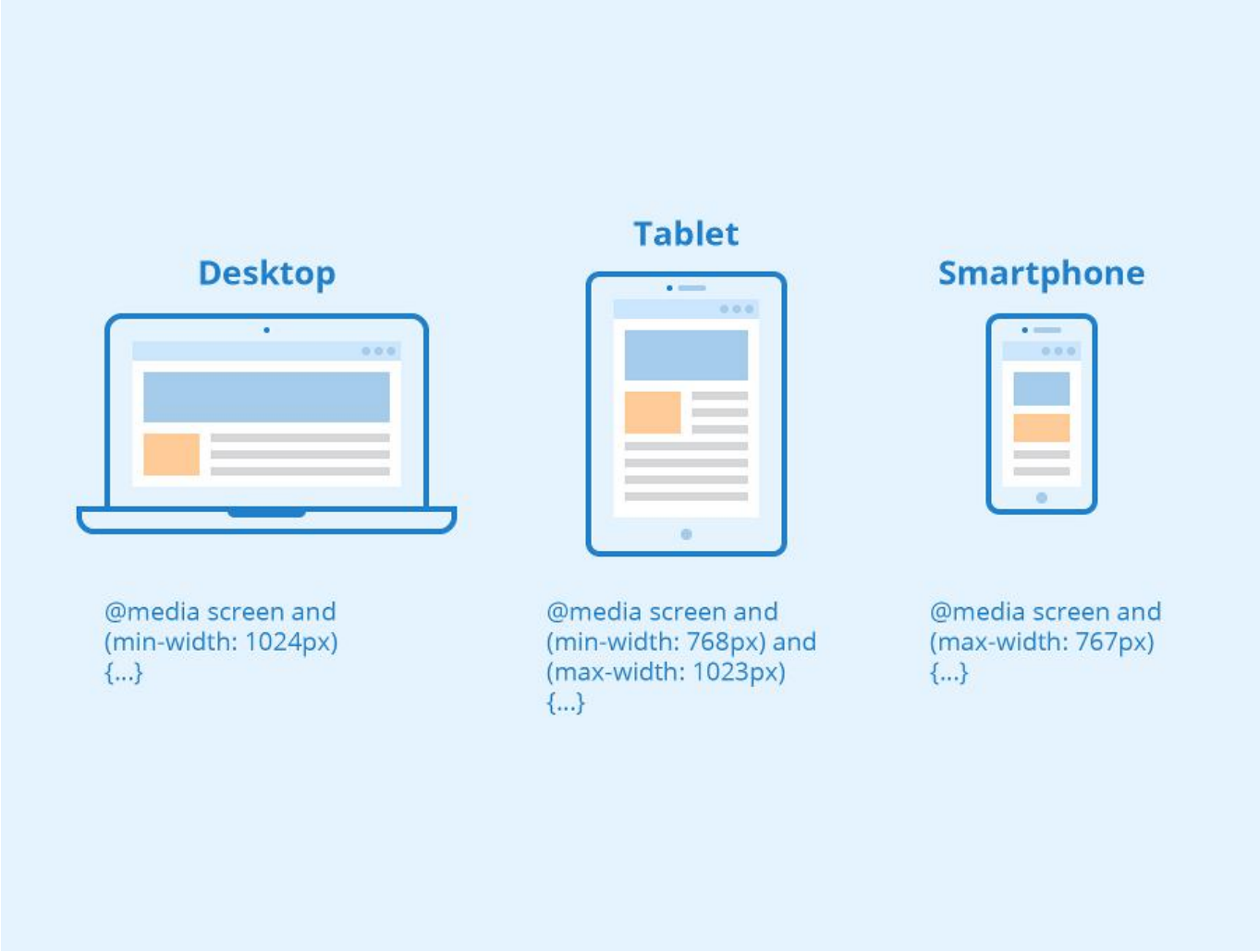
Was passiert?

- Über 1024px gilt: `row`
- Unter 1024px wird die zweite Regel aktiv und überschreibt `flex-direction`

Media Query überschreibt nur dann, wenn sie „an“ ist. Und auch dort gilt: Spezifität + Reihenfolge entscheiden.

> **CSS debuggen**
> Wenn etwas „komisch“ aussieht, schauen Sie in DevTools: Welche Regel ist durchgestrichen? Wer gewinnt? Genau das ist die Kaskade.

Media Queries & Responsive Design



Was macht eine Media Query?

Media Queries erlauben, dass sich das Layout an verschiedene Bildschirmgrössen anpasst, ohne den Inhalt zu ändern – nur die Darstellung wird angepasst.

Breakpoints: nach Inhalt, nicht nach Geräten

Ein **Breakpoint** ist eine Bildschirmbreite, bei der Sie Ihr CSS **bewusst ändern**, damit das Layout wieder gut aussieht. Das passiert meistens über eine **Media Query**, z.B.:

```
@media (width < 1024px) { ... }
```

Was passiert an einem Breakpoint?

Bei einem Breakpoint passen Sie die Darstellung an, weil sonst z.B.:

- Elemente **überlappen** oder sich gegenseitig verdrängen
- Textzeilen extrem lang/kurz werden (schlecht lesbar)
- Buttons/Navigation nicht mehr sinnvoll Platz haben
- ein 2-Spalten-Layout zu eng wird und unruhig wirkt
- horizontales Scrollen entsteht

Ein Breakpoint ist also der Moment, wo das Layout „kippt“ und eine andere Anordnung sinnvoller ist.

Beispiel aus unserem Projekt

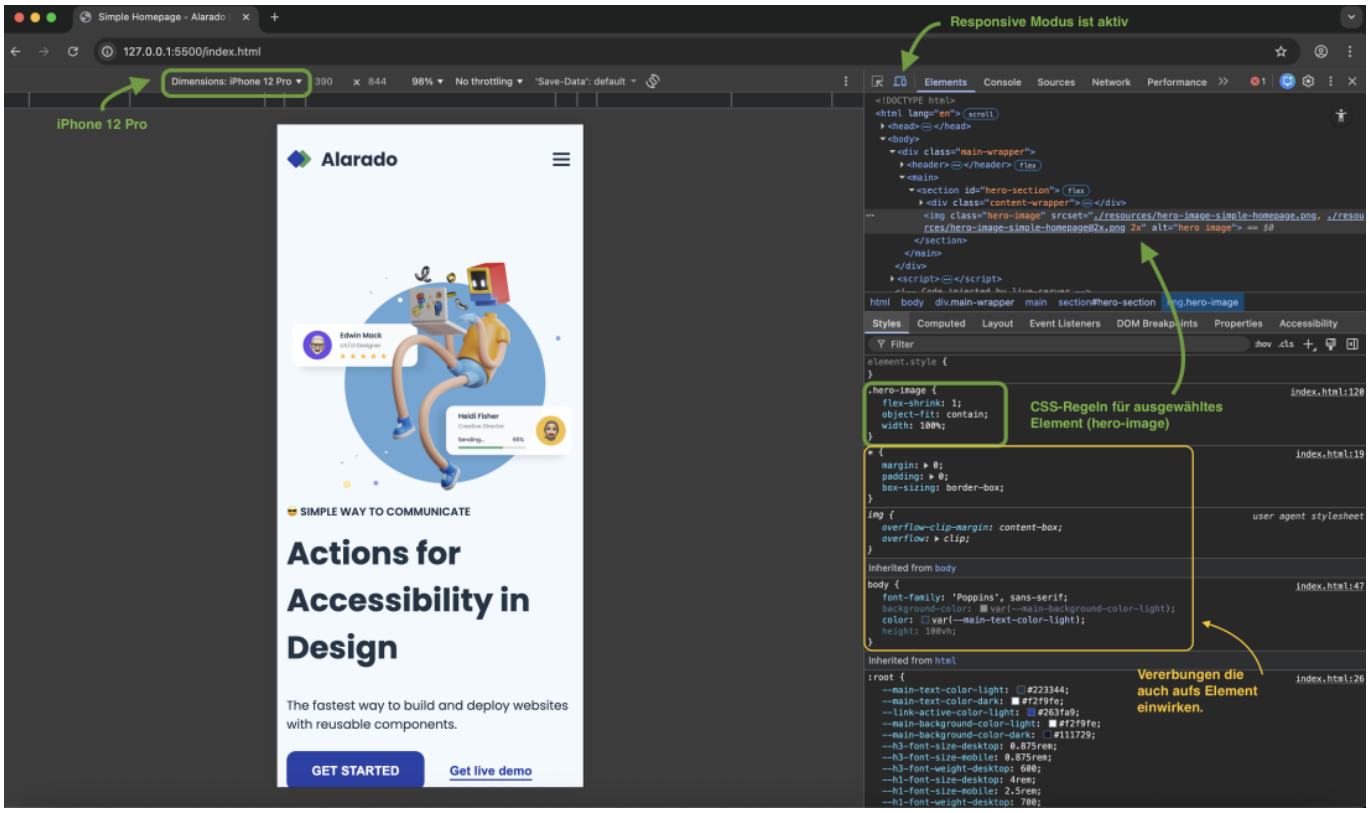
Unser Desktop-Layout ist 2-spaltig (Text links, Bild rechts). Auf Mobile wird es 1-spaltig, mit dem Bild zuerst:

```
@media screen and (width < 1024px) {
  #hero-section {
    flex-direction: column-reverse; /* Bild oben, Text unten */
  }
}
```

Media Queries und Kaskade

- Innerhalb der Media Query überschreiben Sie **nur** die Eigenschaften, die sich ändern sollen.
- Wenn ein Selektor gleich bleibt (z.B. `#hero-section`), gewinnt bei aktiver Media Query die Regel, die **später** kommt (also im `@media`-Block).

Responsive Testing mit DevTools



- Öffnen Sie DevTools (Chrome/Edge: F12)
- Aktivieren Sie den **Device Toolbar** (Handy/Tablet Symbol)
- Testen Sie:
  1. Breitenwechsel über/unter **1024px**
  2. verschiedene Presets (z.B. iPhone, iPad)
  3. Landscape/Portrait
- Prüfen Sie auch:
  1. ob Elemente verschwinden/überlappen
  2. ob Text zu klein/gross wirkt
  3. ob es horizontales Scrollen gibt (meist ein Zeichen für zu grosse Breiten)

1)

Historisch stammt „em“ aus der Typografie (em-square / „M“-Breite). In CSS ist die Definition jedoch eindeutig: 1em entspricht der aktuell berechneten `font-size`-Grösse – nicht der Buchstabenbreite.

2)

In diesem Modul: `!important` nur, wenn Sie genau wissen, warum – sonst vermeiden.