# Projektdokumentation – Schmuck‑Shop

## 1. Einleitung

### 1.1 Projektumfeld

Das Projekt **„Schmuck‑Shop”** wurde im Rahmen des Workshops „Team Fian” durchgeführt. Es diente als praxisnahes Ausbildungsprojekt für angehende Fachinformatiker:innen der Fachrichtung Anwendungsentwicklung. Ein fiktiver Auftraggeber (der Umschulungsträger) wünschte einen prototypischen E‑Commerce‑Shop, um den Teilnehmenden den vollständigen Software‑Entwicklungszyklus anhand eines greifbaren Beispiels zu vermitteln. Als Stakeholder fungierten das Projektteam Team Fian (n = 5), der betreuende Dozent als Auftraggeber sowie der Träger als übergeordnete Instanz. Projektstart war der **9. Oktober 2025**, die Abnahme und Übergabe erfolgten am **20. Oktober 2025**, wodurch sich ein Zeitfenster von 80 Arbeitsstunden ergab.

### 1.2 Projektziel

Die Zielsetzung des Projekts orientierte sich an der SMART‑Methodik. Der Schmuck‑Shop sollte bis **20. Oktober 2025** als funktionsfähiger Prototyp bereitstehen. Das System sollte fünf Kernfunktionen bereitstellen: (1) eine benutzerfreundliche **Produktverwaltung** für Schmuckartikel (Anlegen, Bearbeiten, Löschen) inklusive Bild‑Uploads; (2) einen **Produktkatalog** mit Such‑ und Filterfunktionen sowie einer responsiven Darstellung; (3) einen **Warenkorb** mit Auswahl von Varianten und einer Netto/MwSt‑Berechnung; (4) einen **Bestellvorgang** mit Dummy‑Checkout für Testzwecke; und (5) eine **Benutzer‑ und Entwicklerdokumentation**. Das Ziel war klar messbar (fünf vollständig implementierte Kernfunktionen), spezifisch (Prototyp eines Schmuck‑Shops), akzeptiert (vom Auftraggeber abgenommen), realistisch (80 Stunden Arbeitsumfang) und terminiert (20. Oktober 2025).

### 1.3 Ausgangssituation und Projektbegründung

Im Vorfeld des Projekts existierte kein prototypischer Shop, der den Teilnehmenden die E‑Commerce‑Entwicklung realitätsnah veranschaulicht hätte. Die Umschüler:innen kannten fertige Systeme wie WooCommerce oder Shopify, hatten jedoch kaum Erfahrung in der Entwicklung eigener Lösungen. Der Auftraggeber stellte daher die Aufgabe, einen **prototypischen Schmuck‑Shop** zu entwickeln, um Fachwissen in Frontend‑ und (optional) Backend‑Entwicklung, Datenmodellierung, UX‑Design sowie Dokumentation zu fördern. Das Projekt sollte zudem Grundlagen im Requirements Engineering, in der agilen Planung sowie im Testen vermitteln. Durch die Eigenentwicklung wurde der Lernnutzen maximiert: Teilnehmende müssen sich mit allen Phasen der Softwareentwicklung befassen – von der Analyse über den Entwurf bis zur Implementierung und Qualitätssicherung.

### 1.4 Projektschnittstellen

Der Prototyp wurde als rein **statische Webanwendung** konzipiert. Sämtliche Seiten (index.html, shop.html, checkout.html, register.html, willkommen.html und seitenindex.html) basieren auf **HTML5** und **CSS3**. Für das Styling kam das Framework **Bootstrap 5.3** zum Einsatz, eingebunden über ein Content‑Delivery‑Network (CDN). Die **Geschäftslogik** wurde mit **Vanilla JavaScript** realisiert. Produktdaten werden aus einer lokal vorliegenden JSON‑Datei (assets/data/products.json) geladen. Benutzer‑ und Warenkorbdaten werden clientseitig in der localStorage‑API gespeichert. Die Anwendung benötigt daher weder einen Backend‑Server noch eine Datenbank und ist mit jedem modernen Browser ausführbar.

#### Prozessschnittstellen

Auch organisatorische Schnittstellen wurden definiert. Der Einstieg ins Projekt war die Genehmigung des Projektantrags am **8. Oktober 2025** durch den Auftraggeber. Der Ausstieg bzw. die Übergabe erfolgte am **20. Oktober 2025** im Rahmen einer Präsentation. In regelmäßigen Feedback‑Terminen mit dem Dozenten wurden Anforderungen verifiziert und Abweichungen besprochen. Diese **Prozessschnittstellen** (Kick‑off, Abnahme, wöchentliche Reviews) stellten sicher, dass das Projektteam kontinuierlich Rückmeldung erhielt und der Projektverlauf nachvollziehbar dokumentiert ist.

### 1.5 Projektabgrenzung

Der Fokus lag auf dem Frontend‑Prototyp des Shops. **Nicht Bestandteil** waren ein echter Zahlungsprozess, eine rechtskonforme Bereitstellung im Internet, eine Benutzerverwaltung inklusive Login und Rollen, ein Backend mit Datenbank sowie ein Deployment in die Cloud. Eine einfache Registrierung mit Vor‑ und Nachname, E‑Mail und gehashtem Passwort wurde implementiert, aber kein umfassendes Authentifizierungssystem. Diese Punkte wurden bewusst ausgeklammert, um den Scope innerhalb von 80 Stunden realistisch zu halten. Eine spätere Erweiterung um serverseitige Funktionen ist denkbar und wird im Ausblick skizziert.

## 2. Projektplanung

### 2.1 Meilensteine und Zeitplanung

Zur Strukturierung der 80 Arbeitsstunden wurden acht Meilensteine definiert. Der Projektzeitraum erstreckte sich vom **9. Oktober 2025** bis **20. Oktober 2025** (Wochenenden ausgenommen). Die Tabelle 1 zeigt Meilensteine und Termine:

| Meilenstein | Datum | Beschreibung |
| --- | --- | --- |
| **M1: Kick‑off & Planung** | 09.10.2025 | Projektantrag genehmigt, Projektziel und Scope festgelegt |
| **M2: Anforderungen & Mockups verabschiedet** | 10.10.2025 | Kundengespräche, Use Cases und Mockups erstellt und abgenommen |
| **M3: Technisches Setup abgeschlossen** | 13.10.2025 | Dateien und Ordnerstruktur angelegt, Toolchain eingerichtet |
| **M4: Backend‑MVP** | 15.10.2025 | (Optionales) Flask‑Backend prototypisch vorbereitet (später verworfen) |
| **M5: Frontend‑MVP** | 16.10.2025 | Produktkatalog, Suche, Warenkorb und UI‑Design implementiert |
| **M6: Testphase & Bugfixing abgeschlossen** | 17.10.2025 | Manuelle Tests ausgeführt, Fehler behoben |
| **M7: Dokumentation fertig** | 20.10.2025 | Projekt‑ und Benutzerdokumentation erstellt |
| **M8: Präsentation & Übergabe** | 20.10.2025 | Abnahme durch Dozent, Abschlusspräsentation und Übergabe |

Diese Meilensteine wurden im Projektplaner (Gantt‑Diagramm) abgebildet. Durch **Timeboxing** und tägliche Kurzabstimmungen wurde sichergestellt, dass der Projektfortschritt transparent blieb und rechtzeitig Anpassungen vorgenommen werden konnten. Größere Iterationen (Analyse, Entwurf, Implementierung, Test) wurden durch sogenannte „Sprints” von zwei Tagen unterstützt.

#### 2.1.1 Terminvergleich (Soll vs. Ist)

Alle Meilensteine wurden im geplanten Zeitfenster erreicht. Die nachfolgende Tabelle zeigt den Abgleich zwischen den ursprünglich definierten Terminen und den tatsächlich erreichten Daten. Da der Projektverlauf weitgehend ohne Verzögerungen verlief, stimmen Soll‑ und Ist‑Termine überein; Abweichungen wären hier besonders kenntlich gemacht worden. Dieses Termintracking dient der Nachvollziehbarkeit gegenüber der Prüfungskommission.

| Meilenstein | Geplanter Termin | Tatsächlicher Termin | Abweichung |
| --- | --- | --- | --- |
| **M1: Kick‑off & Planung** | 09.10.2025 | 09.10.2025 | ±0 Tage |
| **M2: Anforderungen & Mockups** | 10.10.2025 | 10.10.2025 | ±0 Tage |
| **M3: Technisches Setup** | 13.10.2025 | 13.10.2025 | ±0 Tage |
| **M4: Backend‑MVP** | 15.10.2025 | — *verworfen* | – |
| **M5: Frontend‑MVP** | 16.10.2025 | 16.10.2025 | ±0 Tage |
| **M6: Testphase & Bugfixing** | 17.10.2025 | 17.10.2025 | ±0 Tage |
| **M7: Dokumentation fertig** | 20.10.2025 | 20.10.2025 | ±0 Tage |
| **M8: Präsentation & Übergabe** | 20.10.2025 | 20.10.2025 | ±0 Tage |

*Hinweis:* Der Meilenstein **M4** sah ursprünglich die Vorbereitung eines minimalen Flask‑Backends vor. Dieser Baustein wurde nach Abstimmung mit dem Auftraggeber aus dem Scope genommen (siehe Abschnitt 3.3); daher gibt es hier keinen Ist‑Termin.

### 2.2 Aufwandsverteilung nach Phasen

Um den Ressourceneinsatz transparenter zu machen, wurde der geplante Aufwand auf die typischen Phasen der Softwareentwicklung verteilt. Dabei handelt es sich um eine Schätzung, die den 80‑Stunden‑Rahmen als Grundlage nimmt. Tabelle 2 zeigt die Aufwandsverteilung:

| Phase | Zeit (h) | Anteil | Beschreibung |
| --- | --- | --- | --- |
| **Analyse & Planung** | 12 | 15 % | Anforderungen erheben, User Stories formulieren, Mockups erstellen |
| **Entwurf** | 8 | 10 % | Struktur des Prototyps festlegen, Datenmodell und UI konzipieren |
| **Implementierung** | 40 | 50 % | Entwicklung der Module (ui.js, products.js, app.js, auth.js), Styling und Integration |
| **Test** | 10 | 12,5 % | Erstellung von Testfällen, manuelle Tests der User Stories, Fehlersuche |
| **Dokumentation** | 10 | 12,5 % | Verfassen der Projekt‑ und Benutzerdokumentation, Auswertung der Ergebnisse |

Die tatsächlichen Zeiten können je nach individueller Arbeitsweise abweichen. Die Übersicht dient primär dazu, den Schwerpunkt der Arbeit zu verdeutlichen (Implementation) und zu zeigen, dass auch für Tests und Dokumentation ausreichend Zeit eingeplant wurde.

### 2.3 Ressourcenplanung

Das Team arbeitete mit **Visual Studio Code**, **Git** und einem modernen Browser (Google Chrome). Als Betriebssystem diente **Windows 11** sowie für Teile Linux. Die einzige benötigte Serverkomponente war ein lokaler HTTP‑Server für das Testen, z. B. via npx http-server. Alle eingesetzten Tools und Bibliotheken sind **Open Source**. Tabelle 2 zeigt eine Übersicht der Ressourcen und angenommenen Kosten:

| Ressource | Menge | Stundensatz | Zeit | Kosten (EUR) | Bemerkungen |
| --- | --- | --- | --- | --- | --- |
| Dozent/Projektleitung | 1 Person | 80 €/Stunde | 80 h | 6 400 € | Feedback, Abnahme, Begleitung |
| Teilnehmende (Team Fian) | 5 Personen | — | 80 h | 0 € | Ausbildung, keine externe Verrechnung |
| Hardware (PCs, Laptops) | — | pauschal | — | 0 € | Bereitgestellt vom Träger |
| Software (VS Code, Git, etc.) | — | Open‑Source | — | 0 € | Keine Lizenzkosten |
| Sonstige Kosten (Material) | — | pauschal | — | 50 € | Ausdrucke, Papier |

**Gesamtkosten (angenommen): 6 450 €**. Da das Projekt ein Ausbildungszweck verfolgte, wurden die Arbeitsstunden der Teilnehmer nicht verrechnet. Die Hauptkosten entstehen durch die projektleitende Person und geringfügige Ausgaben für Material.

### 2.4 Entwicklungsprozess

Der Entwicklungsprozess orientierte sich am **agilen Vorgehensmodell** mit kurzen Iterationen und engmaschigem Feedback. Zunächst wurden in der Analysephase Anforderungen erhoben und priorisiert. Anschließend wurde anhand von Mockups und einem technischen Proof‑of‑Concept entschieden, auf ein eigenständiges Backend zu verzichten und einen rein statischen Shop zu entwickeln. Jede Projektphase (Analyse, Entwurf, Implementierung, Test) wurde in zwei‑tägigen Sprints durchlaufen. Nach jedem Sprint erfolgte eine Review mit dem Auftraggeber, in der die Ergebnisse demonstriert und Anforderungen angepasst wurden. Diese **iterative Vorgehensweise** ermöglichte es, früh funktionsfähige Teilergebnisse zu liefern und Risiken (z. B. Zeitknappheit, technische Hürden) durch frühzeitige Tests zu minimieren.

Um das agile Vorgehen zu stützen, wurden zusätzlich Elemente aus dem **Scrum‑Framework** übernommen: Ein **Product Backlog** hielt alle User Stories und Aufgaben samt Priorität und Akzeptanzkriterien fest. In täglichen **Stand‑up‑Meetings** (ca. 15 Minuten) tauschte sich das Team über Fortschritte, Hindernisse und geplante Schritte aus. Eine **Definition of Ready (DoR)** legte fest, wann eine Anforderung ausreichend beschrieben war, um in einen Sprint aufgenommen zu werden; die **Definition of Done (DoD)** definierte, dass eine Aufgabe erst als erledigt gilt, wenn sie implementiert, getestet, dokumentiert und von einem anderen Teammitglied reviewt wurde. Durch diese Praktiken wurden Transparenz und Qualität im Projektverlauf weiter verbessert.

## 3. Analysephase

### 3.1 Ist‑Analyse

Die Teilnehmenden verfügten über Grundkenntnisse in Webentwicklung (HTML, CSS, JavaScript), hatten jedoch nur begrenzt Erfahrung mit der Entwicklung eines kompletten E‑Commerce‑Systems. Gängige Shop‑Systeme wie Shopify oder WooCommerce waren bekannt, aber der konkrete Aufbau eines Systems mit Routing, Datenmodell, Warenkorb‑Logik und Formularvalidierung war unbekannt. Das Projekt diente daher als „Realitätslabor”, in dem diese Lücke geschlossen werden sollte.

### 3.2 Wirtschaftlichkeitsanalyse

Der Prototyp verfolgte kein wirtschaftliches Ziel; vielmehr stand der Lerngewinn im Vordergrund. Dennoch wurde gegenübergestellt, welche Kosten und Nutzen mit einer Eigenentwicklung verbunden wären. Die Nutzung eines bestehenden Shopsystems hätte weniger Entwicklungsaufwand verursacht, aber auch keine fundierte Ausbildung in Softwareentwicklung ermöglicht. Durch die Eigenentwicklung erwarb das Team Kenntnisse in Frontend‑Entwicklung, Testen, Dokumentation und Projektmanagement, was langfristig die Selbstständigkeit stärkt. Die Kosten beschränkten sich auf die Betreuung durch den Dozenten.

### 3.3 Nutzwertanalyse

Für die initial geplante Backend‑Komponente wurde eine **Nutzwertanalyse** durchgeführt. Verglichen wurden die Frameworks **Flask** (Python), **Django** (Python) und **Express** (Node.js). Kriterien wie Lernkurve, Time‑to‑First‑Feature, Community‑Unterstützung, Flexibilität und E‑Commerce‑Eignung wurden bewertet und gewichtet. Tabelle 3 zeigt die Ergebnisse (Skala 1–5, höher = besser, Gewichtung in Klammern):

| Kriterium | Gewicht | Flask | Django | Express |
| --- | --- | --- | --- | --- |
| Lernkurve | 0,20 | 5 | 3 | 3 |
| Setup/Time‑to‑First‑Feature | 0,20 | 5 | 3 | 4 |
| Community & Ressourcen | 0,15 | 4 | 5 | 5 |
| Flexibilität/Leichtgewicht | 0,15 | 5 | 3 | 4 |
| E‑Commerce‑Eignung (Lehrzweck) | 0,15 | 4 | 5 | 4 |
| Geringe Systemanforderungen | 0,15 | 5 | 3 | 4 |

Flask erzielte die höchste gewichtete Summe und wurde als geeignete Plattform identifiziert. In der Implementierung wurde aufgrund des geringen Zeitbudgets jedoch auf ein Backend verzichtet.

Da im weiteren Projektverlauf vollständig auf eine serverseitige Komponente verzichtet wurde, blieb die obige Analyse ohne konkrete Umsetzung. Stattdessen wurde entschieden, den Shop rein clientseitig mit **Vanilla JavaScript** zu realisieren und kein Framework wie React, Vue oder Angular einzusetzen. Die Wahl fiel auf eine schlanke Lösung, um die Lernziele (DOM‑Manipulation, Event‑Handling und Datenpersistenz) unmittelbar erfahrbar zu machen. Ein späterer Umstieg auf ein Frontend‑Framework bleibt möglich und wird im Ausblick thematisiert.

### 3.4 Risikoanalyse

Im Rahmen der Risikoanalyse wurden potenzielle Gefahren identifiziert, bewertet (Eintrittswahrscheinlichkeit × Auswirkung) und mit Gegenmaßnahmen versehen. Tabelle 4 fasst die wichtigsten Risiken zusammen:

| Risiko | Eintritt | Auswirkung | Risikowert | Gegenmaßnahmen |
| --- | --- | --- | --- | --- |
| **Zeitknappheit (80 Std.)** | Hoch | Mittel | 4 | Strikter Scope, tägliche Stand‑ups, Timeboxing |
| **Technische Hürden (z. B. Stripe)** | Mittel | Mittel | 3 | Frühe Prototypen, Fallback auf Dummy‑Implementierung |
| **Qualitätsmängel** | Mittel | Hoch | 4 | Unit‑Tests, Code‑Reviews, Definition of Done (DoD) |
| **Datenverlust** | Niedrig | Hoch | 3 | Git, häufige Commits, Backups |
| **Teilnehmerausfall** | Mittel | Mittel | 3 | Pair‑Programming, Dokumentation, modulare Aufgaben |
| **Rechtliche Seiten unvollständig** | Mittel | Mittel | 3 | Platzhalter, rechtliche Checkliste, spätere Ergänzung |

### 3.5 Ressourcen‑ & Kostenplan

Der Ressourcen‑ und Kostenplan wurde bereits in Abschnitt 2.2 dargestellt. Die wichtigste Erkenntnis aus der Analysephase war, dass die Entwicklerressourcen kostenneutral sind (da es sich um eine Ausbildungsleistung handelt) und die größten Kosten durch Betreuung und Organisation entstehen.

### 3.6 Anforderungsdokumentation

Die fachlichen Anforderungen wurden als **User Stories** festgehalten und mit eindeutigen IDs, Kategorien (funktional, qualitativ) sowie Prioritäten versehen. Jede Story besitzt Akzeptanzkriterien, anhand derer die Erfüllung überprüft werden kann. Tabelle 5 zeigt einen Auszug der wichtigsten Anforderungen:

| ID | Kategorie | Beschreibung (User Story) | Priorität | Akzeptanzkriterien |
| --- | --- | --- | --- | --- |
| **US01** | funktional | *Als Besucher möchte ich einen Produktkatalog mit Such‑ und Filterfunktion nutzen, damit ich passenden Schmuck finde.* | Hoch | Katalog listet Produkte, Suchfeld filtert nach Name/Tag, Filter nach Kategorie funktioniert, Seiten laden < 2 s |
| **US02** | funktional | *Als Besucher möchte ich Produkte in einen Warenkorb legen und Varianten auswählen können, um mehrere Artikel zu einer Bestellung zusammenzustellen.* | Hoch | Button „In den Warenkorb” vorhanden, Variantenwahl vor Speicherung, Warenkorb zeigt Anzahl, Summe und MwSt, Änderungen möglich |
| **US03** | funktional | *Als Besucher möchte ich meine Bestellung über einen Dummy‑Checkout abschließen können, um den Bestellprozess zu testen.* | Mittel | „Zur Kasse” führt zu Bestellübersicht, Summenberechnung korrekt, Bestätigungsmeldung erscheint |
| **US04** | funktional | *Als Administrator möchte ich neue Produkte anlegen, bearbeiten und löschen können, damit das Sortiment gepflegt werden kann.* | Mittel | CRUD‑Funktionen funktionieren ohne Fehler, Änderungen sofort im Katalog sichtbar |
| **US05** | qualitativ | *Als Benutzer möchte ich den Shop auf Mobilgeräten problemlos nutzen können, damit ich auch unterwegs einkaufen kann.* | Hoch | Layout ist responsiv, keine horizontalen Scrollbalken, Buttons und Texte sind auch bei kleinen Displays lesbar |
| **US06** | qualitativ | *Als Benutzer möchte ich schnelle Ladezeiten (Startseite < 2 s bei 50 Produkten), damit die Bedienung nicht frustrierend ist.* | Mittel | Ladezeiten werden per DevTools gemessen, keine Blockierung durch große Skripte |

Die vollständige Spezifikation enthält weitere Stories, etwa zur Passwortvalidierung, zum Hashing oder zur Modaldarstellung von Produktdetails. Diese Anforderungen wurden mit dem Auftraggeber abgestimmt und im Rahmen der Planung priorisiert. Die Abnahme (siehe Abschnitt 6) prüfte alle Akzeptanzkriterien.

### 3.7 Qualitätsanforderungen

Qualitative Anforderungen ergänzten die funktionalen Stories. Dazu gehörten vor allem:

* **Bedienbarkeit und Zugänglichkeit:** Die Benutzeroberfläche musste intuitiv, barrierearm und responsive gestaltet sein. Kontraste, ausreichend große Buttons und verständliche Fehlermeldungen wurden sichergestellt.
* **Performance:** Ladezeiten sollten niedrig bleiben; Ziel war eine Latenz < 2 s für das Rendern des Katalogs mit 50 Produkten. Durch Lazy Loading und minimierte Assets wurde dies erreicht.
* **Sicherheit:** Passwörter werden clientseitig mittels Web‑Crypto‑API zu SHA‑256 gehasht, bevor sie im localStorage gespeichert werden. Formularfelder werden validiert, um SQL‑Injection‑ähnliche Angriffe zu verhindern (Cross‑Site‑Scripting ist in reinem Frontend jedoch möglich und wird in der Ausblick‑Sektion adressiert).
* **Codequalität:** Es wurden Codestandards (z. B. semantische HTML5‑Tags, modulare JavaScript‑Dateien) eingehalten. Eine DoD definiert, wann eine Aufgabe als abgeschlossen gilt (Review, Tests, Dokumentation). Code‑Reviews und Formatierungstools (Prettier) halfen, die Qualität zu sichern.

### 3.8 Anforderungsherleitung und Validierung

Die Sammlung der fachlichen und qualitativen Anforderungen erfolgte iterativ und transparent in enger Abstimmung mit dem Auftraggeber (Dozent). Zu Projektbeginn führte das Team einen **Anforderungs‑Workshop** durch, in dem gemeinsam mit dem Auftraggeber die Kernfunktionen definiert wurden. Aus diesen Bedürfnissen wurden **User Stories** formuliert und mit **Akzeptanzkriterien** versehen. Die Stories orientierten sich an dem in der Schulung vermittelten Format „Als … möchte ich …, damit …”.

Im Anschluss erstellte das Team **Low‑Fidelity‑Mockups**, die den Katalog, die Produktdetails, den Warenkorb und die Registrierung visualisierten. Diese Entwürfe halfen dabei, eine gemeinsame Vorstellung der Benutzeroberfläche zu entwickeln. Die Mockups wurden im zweiten Meilenstein (M2) dem Auftraggeber vorgestellt und von ihm abgenommen. Änderungswünsche (z. B. eine responsivere Darstellung für Mobilgeräte) flossen in die Spezifikation ein und wurden in den User Stories angepasst.

Die Akzeptanzkriterien jeder Story dienten während der Implementierungs‑ und Testphase als Maßstab. Erst wenn alle Kriterien erfüllt waren, galt eine Anforderung als umgesetzt (siehe Definition of Done in Abschnitt 2.4). Die Validierung erfolgte durch Reviews mit dem Auftraggeber nach jedem Sprint sowie im Rahmen der formellen Abnahme (siehe Abschnitt 6). Durch dieses Vorgehen wird sichergestellt, dass die Anforderungen nicht nur dokumentiert, sondern auch verifiziert und vom Auftraggeber akzeptiert wurden.

## 4. Entwurfsphase

### 4.1 Zielplattform

Die Anwendung läuft auf modernen Webbrowsern. Es gibt keine serverseitige Logik; alle benötigten Daten werden zur Laufzeit in den Browser geladen oder dort gespeichert. Dies hat den Vorteil, dass die Anwendung ohne Installation verwendet werden kann. Gleichzeitig ergeben sich Beschränkungen: Es existiert keine Benutzerverwaltung (kein Login, keine Rollen) und keine echte Zahlungsabwicklung. Der Entwurf berücksichtigt daher eine spätere Integration eines Backends (siehe Ausblick), indem Schnittstellen für den Austausch von Produkten, Benutzern und Bestellungen bereits konzipiert sind (REST‑ähnliche Endpunkte).

### 4.2 Projektstrukturplan und Ablauf

Das Projekt wurde mithilfe eines **Projektstrukturplans (PSP)** in Arbeitspakete unterteilt. Wichtige Pakete waren:

1. **Analyse & Anforderungsdefinition:** Kick‑off, Interviews mit Auftraggeber, Erstellung der User Stories und Mockups.
2. **Technisches Setup:** Anlegen der Ordnerstruktur (assets/js, assets/data, assets/images), Initialisierung des Git‑Repositories, Einbinden von Bootstrap.
3. **Datenmodellierung:** Strukturierung der products.json inklusive aller Attribute (ID, SKU, Name, Kategorie, Beschreibung, Material, Stein, Farbe, Varianten, Bilder, Preis, MwSt, Verfügbarkeit, Tags).
4. **UI‑Entwurf:** Erstellung von Wireframes und Umsetzung mit Bootstrap; Entwicklung von Template‑Strings für Header und Footer (ui.js).
5. **Produktlogik:** Implementierung des Katalogs (app.js), Rendering der Produktkarten, Suche, Filterung, Sortierung, Detailansicht als Modal.
6. **Warenkorb:** LocalStorage‑basiertes Speichern des Warenkorbs, Funktionen zum Hinzufügen, Entfernen, Mengenänderung und Berechnung der Summen.
7. **Registrierung & Auth:** Formularvalidierung, Hashing der Passwörter, Speichern in localStorage (auth.js).
8. **Tests & Qualitätssicherung:** Manuelle Testfälle, ggf. Unit‑Tests mit Jest/Pytest (für den optionalen Backend‑Teil), Code‑Review.
9. **Dokumentation & Präsentation:** Erstellung der Projekt‑ und Benutzerdokumentation, Vorbereitung der Folien.

Der Programmablauf für die Benutzeraktionen wurde in einem **UML‑Aktivitätsdiagramm** dargestellt (im Anhang). Es beschreibt den Ablauf von der Anzeige des Katalogs über die Produktauswahl, Warenkorbverwaltung bis zur Bestellbestätigung. Durch die Visualisierung konnten Abläufe validiert und vereinfacht werden (z. B. Reduktion redundanter Klickpfade).

### 4.3 Benutzeroberfläche

Die UI orientiert sich an modernen E‑Commerce‑Sites: klare Produktkacheln, responsive Grid‑Layouts, Filterleiste und Modal‑Darstellung der Produktdetails. **Bootstrap 5.3** sorgt für Responsiveness. Header und Footer werden als Template‑Strings definiert und per JavaScript in alle Seiten injiziert. Die Darstellung passt sich je nach angemeldeter Person an (z. B. Register‑Button vs. Begrüßung). Schaltflächen verfügen über eindeutige Icons, und Fehlerhinweise werden farblich hervorgehoben. Das **Design** wurde bewusst minimalistisch gehalten, um die Produkte in den Mittelpunkt zu stellen. Für mobile Endgeräte wurde auf ausreichend große Bedienelemente geachtet, und horizontales Scrollen wurde vermieden.

### 4.4 Datenmodell

Der Prototyp nutzt ein JSON‑basiertes Datenmodell. Jedes Produktobjekt besitzt die Felder:

id: fortlaufende numerische ID  
sku: eindeutiger Produktcode (Artikelnummer)  
name: Name des Produkts  
kategorie: Kategorie (Ring, Kette, Armband, Ohrring, ...)  
beschreibung: Beschreibungstext  
material: verwendetes Material  
stein: optionaler Edelstein oder Perle  
farbe: Farbvariante  
varianten: mögliche Größen oder Längen  
bilder: Liste der Bilddateinamen  
preis\_eur: Preis in Euro  
mwst\_satz: Mehrwertsteuersatz (z. B. 0,19)  
verfuegbar: boolean (Lagerbestand > 0)  
tags: Schlagwörter für die Suche

Der Warenkorb wird als Array von Objekten gespeichert. Ein Eintrag besitzt key (Kombination aus SKU und Variante), id, sku, name, price, mwst, variant und qty. Benutzerdaten werden unter dem Schlüssel users abgelegt; jedes Objekt enthält id, firstName, lastName, email, passwordHash und createdAt. Der aktuell angemeldete Nutzer wird unter currentUser gespeichert. Dieses einfache Datenmodell genügt für einen Frontend‑Prototyp und erleichtert eine spätere Migration in eine Datenbank.

### 4.5 Maßnahmen zur Qualitätssicherung

Um die Softwarequalität zu sichern, wurden folgende Maßnahmen umgesetzt:

1. **Testplanung und Testfälle:** Für jede User Story wurden Testfälle definiert, z. B. „Produkt zum Warenkorb hinzufügen” oder „Suche nach Name funktioniert”. Die Testfälle enthielten auch Eingabevarianten und erwartete Ergebnisse.
2. **Manuelle Tests:** Alle Kernfunktionen wurden manuell getestet. Besonderes Augenmerk lag auf der Such‑ und Filterfunktion, dem Warenkorb (Mengenänderung, Entfernen, Summenberechnung), der Registrierung (Validierung, Passwort‑Hashing) sowie auf der Persistenz von Daten über Seiten‑Reloads.
3. **Code‑Review:** Teammitglieder überprüften gegenseitig ihren Code. Die Reviews achteten auf Lesbarkeit, Einhaltung von Konventionen, Modularität und Performance (z. B. Event‑Delegation anstelle mehrerer einzelner Listener).
4. **DoD (Definition of Done):** Eine Aufgabe galt erst als erledigt, wenn sie implementiert, getestet, dokumentiert und von mindestens einer anderen Person reviewt wurde.
5. **Versionsverwaltung:** Alle Fortschritte wurden in Git mit verständlichen Commits festgehalten. Tags markieren die Meilensteine, sodass bei Bedarf ein früherer Stand reproduzierbar ist.

## 5. Implementierungsphase

Die Implementierung erfolgte iterativ und orientierte sich an den Arbeitspaketen des PSP. Nachfolgend werden die wichtigsten Schritte zusammengefasst:

1. **Projektinitialisierung & Dateistruktur:** Erstellung der Basis‑HTML‑Seiten (index.html, shop.html, checkout.html, register.html, willkommen.html, seitenindex.html) sowie des Verzeichnisses assets mit Unterordnern js, data und images. Einrichtung des Git‑Repositories.
2. **Produktdaten:** Erstellung der Datei products.json mit 12 Schmuckartikeln. Für jedes Produkt wurden Name, Kategorie, Beschreibung, Material, Stein, Farbe, Varianten, Preis, MwSt‑Satz, Verfügbarkeit und Schlagwörter festgelegt. Fehlerhafte Bildpfade wurden korrigiert (z. B. .pn → .png).
3. **UI‑Komponenten (**ui.js**):** Implementierung von Header und Footer als Template‑Strings. Dynamischer Austausch der Navigationslinks je nach Login‑Status. Implementierung einer Logout‑Funktion, die currentUser löscht und die Seite neu lädt.
4. **Produktlogik & Katalog (**products.js **&** app.js**):** Laden der Produktdaten via fetch(). Rendering des Katalogs als responsive Grid. Implementierung von Suche (Case‑Insensitive), Filterung nach Kategorie und Sortierung (Preis, Name). Detailansicht über ein Bootstrap‑Modal mit Variantenauswahl. Einführung einer delegierten Ereignisbehandlung auf dem Container (productGrid.addEventListener('click', …)), um Performance zu verbessern.
5. **Warenkorb:** Implementierung eines Warenkorbs als localStorage‑Array (cart). Funktionen zum Hinzufügen, Entfernen und Ändern der Menge. Berechnung der Netto‑ und MwSt‑Summen. Visuelle Rückmeldung nach dem Hinzufügen („Hinzugefügt” → Timeout → „In den Warenkorb”).
6. **Registrierung und Authentifizierung (**auth.js**):** Validierung der Formularfelder (z. B. E‑Mail‑Format mittels RegEx, Passwortlänge und ‑stärke). Hashen der Passwörter mit der Web‑Crypto‑API (crypto.subtle.digest('SHA-256', …)). Speicherung der Benutzerobjekte in localStorage (users). Setzen des currentUser nach erfolgreicher Registrierung und Weiterleitung in den Shop.
7. **Styling:** Anpassung der Bootstrap‑Standardfarben und Typografie über style.css und theme.css. Sicherstellung der Responsiveness. Modale Buttons, Filterelemente und Fehlermeldungen wurden einheitlich gestaltet.
8. **Fehlerbehebungen & Workarounds:** Im Verlauf wurden verschiedene Fehler entdeckt und behoben. Beispiele: Bei leerem localStorage wurde JSON.parse(null) abgefangen; das Verhindern der Registrierung ohne AGB‑Checkbox; temporäre Deaktivierung des „In den Warenkorb”-Buttons nach Klick; Pflege korrekter Pfade zu Bildern; Verbesserung der Event‑Delegation.

## 6. Test‑ & Abnahmephase

Alle User Stories wurden mit definierten Testfällen verifiziert. Da der Prototyp rein clientseitig war, kamen vor allem **manuelle Tests** zum Einsatz:

* **Funktionstests:** Suchen, Filtern, Sortieren, Variantenwahl, Warenkorboperationen, Registrierung. Dabei wurden auch Grenzfälle getestet (leere Suche, Mehrfachhinzufügen, Änderung der Menge auf 0 → Entfernen). Die korrekte Berechnung von Netto‑ und MwSt‑Beträgen wurde mit Testdaten überprüft.
* **Usability‑Tests:** Nutzung auf Desktop und Mobile, Lesbarkeit der Texte, Bedienung der Buttons. Mehrere Teilnehmende testeten das System auf unterschiedlichen Geräten.
* **Datenspeicherung:** Überprüfung, ob Warenkorb‑ und Benutzerdaten nach Seiten‑Reload erhalten bleiben und ob die Registrierung nach dem Schließen des Browsers persistiert ist.

Nach Abschluss der Tests wurden die Ergebnisse mit den Akzeptanzkriterien der User Stories abgeglichen. Der Dozent prüfte in einer formellen **Abnahme** die Erfüllung aller Anforderungen. Das Abnahmeprotokoll dokumentiert, welche Stories vollständig umgesetzt wurden, welche nur teilweise erfüllt sind und welche bewusst aus dem Scope genommen wurden. Ein Auszug des Protokolls ist im **Anhang A4** beigefügt. Die vollständige Testfallliste mit erwarteten und erzielten Ergebnissen findet sich ebenfalls im Repository.

Im Rahmen der Abnahme am **20. Oktober 2025** demonstrierte das Team die Erfüllung aller Akzeptanzkriterien gegenüber dem Dozenten. Ein Abnahmeprotokoll wurde unterzeichnet. Alle geplanten Funktionen funktionierten. Kleinere Anpassungen (UI‑Optimierungen) wurden bis zur Übergabe eingepflegt. Eine exemplarische Testfallliste und das Abnahmeprotokoll befinden sich im Anhang.

## 7. Einführungsphase

Da der Prototyp statisch ist, besteht die Einführungsphase im lokalen Ausrollen der Anwendung. Über ein README wird beschrieben, wie ein *http‑server* gestartet wird (npx http-server) und wie die Seite alternativ per Doppelklick auf index.html geöffnet werden kann. Da kein Backend involviert ist, entfällt eine Installation. Eine produktive Bereitstellung im Internet (z. B. über GitHub Pages) wurde nicht durchgeführt; dies wäre Teil einer möglichen Erweiterung.

## 8. Dokumentation

Drei Dokumente wurden erstellt:

1. **Projektdokumentation:** Dieses Dokument beschreibt den Projektverlauf, die Anforderungen, den Entwurf, die Umsetzung und die Ergebnisse. Es dient der Nachvollziehbarkeit und als Grundlage für die Bewertung durch die Prüfungskommission.
2. **Benutzerdokumentation:** Eine kurze Anleitung erklärt Endanwendern, wie sie den Shop starten (lokaler Server), Produkte suchen, filtern, in den Warenkorb legen, zur Kasse gehen und sich registrieren können. Hinweise zur Passwortwahl und zum Umgang mit der Demo‑Checkout‑Funktion werden gegeben.
3. **Entwickler‑README:** Diese Datei erklärt die Installation der Abhängigkeiten, das Starten des lokalen Servers, den Aufbau der Verzeichnisse sowie die wichtigsten Skriptdateien. Sie richtet sich an Entwicklerinnen und Entwickler, die den Code weiterentwickeln möchten.

## 9. Fazit

### 9.1 Soll‑/Ist‑Vergleich

Alle definierten Kernfunktionen wurden termingerecht umgesetzt. Die Meilensteine wurden eingehalten. Eine Übersicht des Soll‑/Ist‑Abgleichs je Hauptanforderung:

| Anforderung | Geplant | Realisiert | Bemerkungen |
| --- | --- | --- | --- |
| Produktkatalog | ✅ | ✅ | Suche, Filter, Sortierung, responsives Grid |
| Warenkorb & Dummy‑Checkout | ✅ | ✅ | Hinzufügen, Entfernen, Mengenänderung, Summenberechnung |
| Produktverwaltung (Admin‑CRUD) | ✅ | 🔸 | Grundlegend implementiert; UI für Admin noch rudimentär |
| Registrierung & Hashing | ✅ | ✅ | Validierung, Passwort‑Hashing, Persistenz im localStorage |
| Benutzerverwaltung (Login) | ❌ | ❌ | Bewusst ausgeklammert, da Scope‑Limitierung |
| Backend‑Anbindung | ❌ | ❌ | Im Soll als optional vorgesehen, wurde zugunsten eines reinen Frontend‑Prototyps gestrichen |
| Dokumentation | ✅ | ✅ | Projekt‑ und Benutzerdokumentation erstellt |

🔸 = teilweise umgesetzt; ✅ = vollständig umgesetzt; ❌ = nicht Bestandteil.

### 9.2 Lessons Learned

Das Projekt hat gezeigt, wie wichtig eine **strukturierte Planung** und eine konsequente **Scope‑Definition** sind. Durch das Festlegen klarer Ziele und Prioritäten konnte das Team innerhalb der knappen 80 Stunden einen funktionsfähigen Prototyp realisieren. Die Wahl eines rein clientseitigen Ansatzes erwies sich als richtig, da die Komplexität des Backends den Rahmen gesprengt hätte. Im Verlauf wurde deutlich, dass **regelmäßige Reviews** und **Code‑Reviews** wesentlich zur Qualität beitragen. Die Nutzung von **Bootstrap** erleichterte das responsive Design, zeigte aber auch Grenzen hinsichtlich Individualisierung. **Event‑Delegation** im JavaScript erwies sich als performante Lösung gegenüber zahlreichen Einzel‑Listenern. Außerdem wurde klar, dass eine rein Frontend‑basierte Lösung schnell an Grenzen stößt, sobald Benutzerverwaltung, Login oder echte Zahlungen gefordert werden.

### 9.3 Ausblick

Für eine Weiterentwicklung des Schmuck‑Shops bieten sich folgende Schritte an:

1. **Backend‑Integration:** Anbindung an ein Framework wie Flask oder Django, um Benutzerkonten, Bestellungen und Zahlungen serverseitig zu verwalten. REST‑APIs könnten die Frontend‑Komponenten versorgen. Authentifizierung mittels Sessions oder Tokens und die Integration eines Zahlungsdienstleisters (z. B. Stripe, PayPal) würden den Prototyp zum echten Shop erweitern.
2. **Benutzerverwaltung:** Implementierung eines Logins sowie eines Admin‑Dashboards zur Produktpflege und Auftragsverwaltung. Damit ließe sich US04 (CRUD) vollständig umsetzen.
3. **Rechtliche Konformität:** Ergänzung eines vollständigen Impressums, einer Datenschutzerklärung und einer AGB‑Seite. Bei einem realen Deployment sind diese Pflicht.
4. **Design‑Optimierung:** Erstellung eines individuellen Themes, z. B. mit Tailwind CSS oder eigenem SCSS‑Workflow, um sich von Bootstrap‑Standardlayouts abzuheben.
5. **Automatisierte Tests:** Einführung von Unit‑Tests (Jest für JavaScript) und End‑to‑End‑Tests (z. B. Cypress), um die Qualität nachhaltig zu sichern.

Mit diesen Erweiterungen ließe sich der Prototyp zu einem produktiven Webshop ausbauen und gleichzeitig die Lerninhalte vertiefen.

## Anhang

### A1. Glossar (vollständig)

Das Glossar erläutert die wichtigsten Begriffe, Abkürzungen und Konzepte, die in dieser Dokumentation verwendet werden. Bei der ersten Erwähnung eines Fachbegriffs wird auf diesen Abschnitt verwiesen, damit Leserinnen und Leser die Bedeutung schnell nachschlagen können.

| Begriff | Definition / Bedeutung im Projektkontext |
| --- | --- |
| **Backend** | Serverseitige Komponenten, die Daten verarbeiten, speichern und Geschäftslogik bereitstellen. Im Projekt wurde bewusst auf ein Backend verzichtet, um den Fokus auf das Frontend zu legen. |
| **Bootstrap** | CSS‑Framework zur schnellen Gestaltung responsiver Webseiten. Version 5.3 wurde eingesetzt, um das Grid‑Layout, Modale und die Navigation des Shops umzusetzen. |
| **CRUD** | Abkürzung für *Create, Read, Update, Delete* – Grundoperationen zur Datenverwaltung. Die Produktverwaltung soll CRUD‑Funktionen bereitstellen. |
| **LocalStorage** | Browser‑API zum dauerhaften Speichern von Schlüssel‑Wert‑Daten im Client. Der Warenkorb und Benutzerdaten werden im localStorage abgelegt, sodass sie nach einem Seitenreload erhalten bleiben. |
| **User Story** | Beschreibung einer Anforderung aus Sicht eines Benutzers oder Stakeholders im Format „Als … möchte ich …, damit …“. Jede User Story besitzt Akzeptanzkriterien zur Überprüfung ihrer Erfüllung. |
| **IHK** | Abkürzung für *Industrie‑ und Handelskammer*. Sie ist in vielen Ausbildungsberufen Prüfungsträgerin und gibt Leitfäden für Projektdokumentationen vor. |
| **Lastenheft** | Dokument, das die vom Auftraggeber formulierten Anforderungen und Ziele beschreibt („Was“ wird benötigt?). Es bildet die Grundlage für das Pflichtenheft. |
| **Pflichtenheft** | Dokument, das vom Auftragnehmer erstellt wird und beschreibt, wie (mit welchen Mitteln und Funktionen) die Anforderungen aus dem Lastenheft umgesetzt werden sollen. |
| **PSP (Projektstrukturplan)** | Hierarchische Darstellung aller Arbeitspakete eines Projekts. Der PSP dient der Strukturierung und Planung und ist Grundlage des Gantt‑Diagramms. |
| **Gantt‑Diagramm** | Balkendiagramm, das die zeitliche Planung eines Projekts visualisiert. Es zeigt Start‑ und Endzeitpunkt sowie Dauer der einzelnen Arbeitspakete. |
| **DoD (Definition of Done)** | Liste von Kriterien, die erfüllt sein müssen, damit eine Aufgabe als abgeschlossen gilt (z. B. implementiert, getestet, dokumentiert und reviewed). |
| **DoR (Definition of Ready)** | Kriterien, die eine Anforderung erfüllen muss, damit sie in einen Sprint aufgenommen werden kann. |
| **Mockup** | Entwurf oder Skizze der Benutzeroberfläche. Mockups wurden im Projekt genutzt, um frühzeitig das Layout des Shops zu visualisieren. |
| **Sprint** | Kurze Iteration in agilen Vorgehensmodellen (z. B. Scrum). Im Projekt wurden zwei‑tägige Sprints verwendet, um Analyse, Entwurf, Implementierung und Test in kleinen Schritten durchzuführen. |
| **Product Backlog** | Geordnete Liste aller Anforderungen (User Stories) und Aufgaben eines Projekts, die vom Product Owner bzw. Auftraggeber priorisiert wird. |
| **JSON (JavaScript Object Notation)** | Textbasiertes Format zum Austausch strukturierter Daten. Die Produktdaten werden in der Datei products.json gespeichert. |
| **Event‑Delegation** | Technik in JavaScript, bei der ein einzelner Event‑Listener für mehrere ähnliche Elemente auf einem Container registriert wird. Dadurch wird die Performance verbessert und der Code vereinfacht. |
| **Local HTTP‑Server** | Ein einfacher Webserver (z. B. python3 -m http.server), der lokale Dateien über HTTP bereitstellt. Da viele Browser keine JSON‑Dateien über das Datei‑Protokoll laden dürfen, wird der Shop lokal über einen solchen Server gestartet. |
| **REST** | *Representational State Transfer* – Architekturstil für Web‑APIs. Eine spätere Erweiterung des Shops um ein Backend könnte REST‑Schnittstellen nutzen. |
| **SHA‑256** | Kryptografische Hash‑Funktion, die im Projekt verwendet wird, um Passwörter vor dem Speichern im localStorage zu hashen. |
| **Vanilla JavaScript** | Bezeichnet die Verwendung von JavaScript ohne zusätzliche Frameworks wie React, Vue oder Angular. Im Projekt wurde Vanilla‑JS eingesetzt, um die Lernziele (DOM‑Manipulation, Event‑Handling, LocalStorage) zu vertiefen. |

### A2. Detaillierte Meilensteinbeschreibungen

Für jeden Meilenstein wurde ein kurzes Protokoll erstellt, das die erreichten Arbeitspakete dokumentiert und von Dozent und Teammitgliedern unterschrieben wurde. Ein Beispiel für **M5 (Frontend‑MVP)**:

* Erstellte Module: ui.js, products.js, app.js, auth.js.
* Funktionen: Rendering des Produktkatalogs, Suche, Filter, Sortierung, Warenkorb, Registrierung.
* Probleme: Falsche Bildpfade, fehlendes Event‑Delegation; wurden korrigiert.
* Nächste Schritte: Testen der Kernprozesse, UI‑Feinschliff.

### A3. Screenshots der Anwendung

1. **Startseite / Produktkatalog:** Der Screenshot zeigt den responsiven Katalog mit Suchfeld und Filterleiste. Produkte werden als Kacheln mit Bild, Name, Preis und Button dargestellt.
2. **Produktdetail‑Modal:** Anzeige von Bildern, Beschreibung, Variantenwahl und „In den Warenkorb”-Button innerhalb eines Bootstrap‑Modals.
3. **Warenkorb‑Seite:** Übersicht über alle Artikel inklusive Mengenänderung, Netto‑ und MwSt‑Berechnung sowie Dummy‑Checkout.
4. **Registrierungsseite:** Formular mit Validierung für Vor‑ und Nachname, E‑Mail und Passwort (inkl. Passwortstärke‑Check).

Die Screenshots sind im Projektverzeichnis unter screenshots/ abgelegt und illustrieren den finalen Stand der Benutzeroberfläche.

### A4. Abnahmeprotokoll (Auszug)

Die folgende Tabelle bildet einen Ausschnitt des Abnahmeprotokolls ab. Für jede User Story wird festgehalten, ob alle Akzeptanzkriterien erfüllt wurden. Ein ✅ steht für vollständig erfüllt, 🔸 für teilweise erfüllt, und ❌ für nicht umgesetzt.

| User Story | Beschreibung | Akzeptanzkriterien – Status | Bemerkungen |
| --- | --- | --- | --- |
| **US01** | Produktkatalog mit Suche und Filtern | ✅ – Katalog, Suche & Filter funktionieren, Ladezeit < 2 s | Vollständig umgesetzt |
| **US02** | Warenkorb mit Variantenwahl | ✅ – Warenkorb‑Button, Variantenauswahl, Summenberechnung | Vollständig umgesetzt |
| **US03** | Dummy‑Checkout | ✅ – Bestellübersicht, Summenberechnung, Bestätigung | Vollständig umgesetzt |
| **US04** | Produktverwaltung (CRUD) | 🔸 – Basisfunktionen vorhanden, Admin‑UI rudimentär | Erweiterungen im Ausblick geplant |
| **US05** | Responsives Design | ✅ – Layout passt sich an, gute Lesbarkeit | Vollständig umgesetzt |
| **US06** | Schnelle Ladezeiten | ✅ – DevTools‑Messungen bestätigen Zeiten < 2 s | Durch Lazy Loading erreicht |
| Weitere | Passwortvalidierung, Hashing, Modaldarstellung u.a. | ✅ – Validierung und Hashing umgesetzt | Details im Code einsehbar |

Die vollständige Abnahme wurde am **20. Oktober 2025** vom Dozenten und den Teammitgliedern unterzeichnet. Das komplette Protokoll befindet sich im Projektrepository.

### A5. Code‑Auszug (Beispiel)

Im Folgenden ist ein exemplarischer Ausschnitt aus der Datei app.js dargestellt. Dieser zeigt die delegierte Ereignisbehandlung für das Öffnen des Produktdetail‑Modals sowie die addToCart‑Funktion mit Persistenz im localStorage. Hervorgehobene Codeblöcke werden im Text referenziert (Kapitel 5, Implementierungsphase).

// --- addToCart: fügt ein Produkt inkl. Variante in den Warenkorb ein  
function addToCart(prod, variant){  
 const cart = getCart();  
 const key = prod.sku + '|' + (variant || '');  
 const found = cart.find(i => i.key === key);  
 if(found){  
 found.qty += 1;  
 } else {  
 cart.push({ key, id: prod.id, sku: prod.sku, name: prod.name,  
 price: prod.preis\_eur, mwst: prod.mwst\_satz, variant: variant || '', qty: 1 });  
 }  
 saveCart(cart);  
}  
  
// Delegierter Click‑Handler für Produktkacheln  
const grid = document.getElementById('productGrid');  
if(!grid.\_\_detailsWired){  
 grid.addEventListener('click', function onGridClick(ev){  
 const detailEl = ev.target.closest('[data-detail]');  
 if(detailEl){  
 ev.preventDefault();  
 const sku = detailEl.getAttribute('data-detail');  
 const prod = (window.PRODUCTS || []).find(p => p.sku === sku);  
 if(prod){ openProductModal(prod); }  
 }  
 });  
 grid.\_\_detailsWired = true;  
}

Dieser Code demonstriert die Verwendung der localStorage‑API zur Persistenz des Warenkorbs und die Nutzung eines delegierten Event Listeners zur Performance‑Optimierung. Weitere Codeauszüge (z. B. Registrierung und Hashing) befinden sich im Repository.

### A6. Lasten‑ und Pflichtenheft (Auszug)

Dieser Abschnitt fasst die wichtigsten Inhalte aus Lasten‑ und Pflichtenheft zusammen. Das Lastenheft beschreibt die Ziele und Anforderungen aus Sicht des Auftraggebers, während das Pflichtenheft festlegt, wie diese Anforderungen umgesetzt werden.

**Auszug aus dem Lastenheft:**

* Der Auftraggeber (Bildungsträger) benötigte ein praxisnahes Projekt, um den Software‑Entwicklungszyklus im Rahmen eines Umschulungslehrgangs erlebbar zu machen. Ein prototypischer Online‑Shop für Schmuck sollte diese Lücke schließen.
* Kursteilnehmende verfügten über Grundlagen in HTML, CSS und JavaScript, hatten aber wenig Erfahrung mit der Entwicklung eigener Shop‑Anwendungen. Das Projekt diente dazu, diese Kompetenzlücke zu schließen.
* Kundenwünsche umfassten eine einfache **Produktverwaltung** (Anlegen, Bearbeiten, Löschen), einen übersichtlichen **Produktkatalog** mit Such‑ und Filterfunktionen, einen **Warenkorb** mit Variantenwahl und Dummy‑Checkout, **Responsive Design** für alle Geräte, eine **einfache Bereitstellung** ohne Backend sowie eine verständliche Projekt‑ und Benutzerdokumentation. Der Prototyp sollte bis zum **20. Oktober 2025** innerhalb von 80 Stunden fertiggestellt sein.

**Auszug aus dem Pflichtenheft:**

* Die Produktdaten werden in einer JSON‑Datei (products.json) verwaltet und im Frontend als responsives Grid mit Such‑, Filter‑ und Sortierfunktionen gerendert. Das Laden und Rendern von 50 Produkten darf nicht länger als 2 Sekunden dauern.
* Die Warenkorb‑Funktion speichert Artikel (inkl. Variante und Menge) im localStorage. Ein Modal ermöglicht die Variantenauswahl. Ein Dummy‑Checkout führt zu einer Bestellübersicht und bestätigt den Abschluss (es werden keine echten Daten übertragen).
* Passwörter werden mittels **SHA‑256** gehasht und zusammen mit Benutzerinformationen im localStorage gespeichert. Eine rudimentäre Admin‑UI zur Produktpflege erfolgt über die JSON‑Datei; eine spätere Backend‑Integration ist vorgesehen.
* Zu den nicht‑funktionalen Anforderungen gehören kurze Ladezeiten, responsives Layout, intuitive Bedienung, saubere Codequalität (Einhaltung von DoR/DoD) und die Einhaltung des 80‑Stunden‑Zeitrahmens. Risiken wie Zeitknappheit, technische Hürden oder Qualitätsmängel wurden analysiert und Gegenmaßnahmen definiert.
* Geliefert werden Quellcode, Projekt‑ und Benutzerdokumentation, Fehler‑ und Patch‑Doku sowie eine Präsentation zur Abnahme.

### A7. Projektstrukturplan (PSP) und Gantt‑Diagramm

Der Projektstrukturplan listet alle Arbeitspakete des Projekts hierarchisch auf. Die Stundenangaben beziehen sich auf den vorgesehenen Zeitrahmen von 80 Stunden. Das zugehörige Gantt‑Diagramm visualisiert die zeitliche Abfolge der Pakete (Analyse/Planung, Entwurf, Implementierung, Test, Dokumentation) zwischen dem 9. und 20. Oktober 2025.

| Ebene | Arbeitspaket | Beschreibung | Geplanter Aufwand |
| --- | --- | --- | --- |
| 1 | **Analyse & Requirements Engineering** | Kick‑off, SMART‑Zieldefinition, Erfassen von User Stories und Akzeptanzkriterien, Erstellung der Ressourcen‑ und Risikoanalyse. | 12 h |
| 2.1 | **UX‑ & Designkonzept** | Erstellung eines Navigationskonzepts, Wireframes und Farbschemas. Definition der Usability‑Prinzipien und Auswahl von Bootstrap. | 8 h |
| 2.2 | **Implementierung** | Entwicklung der HTML‑ und CSS‑Strukturen, Implementierung der JavaScript‑Module, Integration der JSON‑Daten, Umsetzung des Warenkorbs, der Registrierung und der Sortier‑/Filterfunktionen. | 40 h |
| 2.2.1 | Produkterfassung & Darstellung | Laden und Anzeigen der Produkte im Grid, Such‑ und Filterlogik, Sortierung nach Name/Preis. | 10 h |
| 2.2.2 | Produktdetails & Modal | Dynamische Generierung des Modals für Produktdetails, Variantenverwaltung und Hinzufügen zum Warenkorb. | 5 h |
| 2.2.3 | Warenkorb & Local‑Storage | Aufbau der Warenkorb‑API (getCart, addToCart, removeFromCart, saveCart), Anzeige der Anzahl und Bearbeitung der Mengen. | 8 h |
| 2.2.4 | Benutzerverwaltung | Registrierung mit Validierung und Passwort‑Hashing, Login‑Logik sowie persistenter Benutzerspeicher. | 10 h |
| 2.2.5 | Benutzeroberfläche & Navigation | Umsetzung der Navigation (Menü, Dropdowns), Footer, Breadcrumbs, responsive Gestaltung. | 3 h |
| 3 | **Test & Qualitätssicherung** | Erstellung von Testfällen für jede User Story, Durchführung der Tests (Funktionalität, Usability, Performance), Fehlererfassung und Behebung. | 10 h |
| 4 | **Dokumentation & Präsentation** | Erstellung der Projekt‑ und Benutzerdokumentation, Fehler‑ und Patch‑Dokumentation, Slides für die Abschlusspräsentation. | 10 h |
| 5 | **Abnahme & Projektabschluss** | Durchführung der Abnahme durch den Auftraggeber, Erstellung des Abnahmeprotokolls, Abschlussbericht und Lessons Learned. | 2 h |

Das Gantt‑Diagramm, das diese Pakete auf der Zeitachse abbildet, befindet sich in der Datei gantt\_chart.png und kann bei Bedarf im Projektordner eingesehen werden.

### A8. Testfallliste (Auszug)

Im Rahmen der Qualitätssicherung wurde für jede User Story eine Reihe von Testfällen definiert. Die folgende Tabelle zeigt einen Auszug der wichtigsten Tests. Weitere Edge‑Cases und Detailtests sind im Projektrepository dokumentiert.

| Test ID | User Story | Testbeschreibung | Erwartetes Ergebnis |
| --- | --- | --- | --- |
| **TC01.01** | US01 – Produktkatalog | Suche nach vorhandenem Produkt (z. B. „Ring“) | Nur Produkte mit dem Suchbegriff werden angezeigt. |
| **TC01.02** | US01 – Produktkatalog | Filter nach Kategorie „Kette“ | Ergebnisliste enthält ausschließlich Produkte der Kategorie „Kette“. |
| **TC01.03** | US01 – Produktkatalog | Sortierung nach Preis absteigend | Produkte werden vom höchsten zum niedrigsten Preis sortiert. |
| **TC02.01** | US02 – Warenkorb | Produkt mit Variante hinzufügen | Warenkorb‑Badge erhöht sich um eins; im Warenkorb erscheint das Produkt mit der gewählten Variante. |
| **TC02.02** | US02 – Warenkorb | Menge ändern | Gesamtpreis passt sich an; Anzahl ändert sich im Warenkorb‑Badge. |
| **TC03.01** | US03 – Dummy‑Checkout | Checkout durchführen | Bestätigungsmeldung erscheint; Warenkorb wird geleert; Local‑Storage bleibt intakt. |
| **TC04.01** | US04 – Produktverwaltung | Neues Produkt in products.json anlegen | Nach dem Laden erscheint das neue Produkt korrekt im Katalog. |

### A9. Fehler‑ und Workaround‑Dokumentation

Im Entwicklungsverlauf wurden verschiedene Fehler identifiziert und behoben. Einige Beispiele sind nachfolgend aufgeführt. Die vollständige Liste findet sich im Fehlerprotokoll des Repositories.

| Nr. | Fehler / Problem | Ursache | Lösung / Workaround |
| --- | --- | --- | --- |
| **1** | Produktmodal öffnet sich nicht beim Klick auf eine Produktkachel | Event‑Listener wurde bei jedem Rendern neu registriert, was den Handler überschrieben hat. | Registrierung des Click‑Handlers außerhalb der Render‑Funktion und Verwendung eines Flags (\_\_detailsWired), um eine doppelte Registrierung zu vermeiden. |
| **2** | Warenkorb leert sich nach Seitenreload | Warenkorb wurde initial im sessionStorage gespeichert, dessen Inhalt beim Schließen des Tabs verloren geht. | Persistenter Warenkorb im localStorage; beim Laden werden die Daten aus dem localStorage eingelesen. |
| **3** | Registrierung akzeptiert unsichere Passwörter | Fehlende Passwort‑Validierung. | Einführung einer RegEx‑Prüfung (Mindestlänge, Groß‑/Kleinschreibung, Zahl, Sonderzeichen) und Anzeige eines Fehlers bei falscher Eingabe. |
| **4** | Browser lädt keine JSON‑Daten (insb. Firefox) | Sicherheitsbeschränkungen verhindern das Laden lokaler JSON‑Dateien über file://. | Starten eines lokalen HTTP‑Servers (z. B. python3 -m http.server) oder Nutzung des Live‑Servers im Editor. |

Zusätzlich wurden Workarounds und Verbesserungen implementiert, z. B. Event‑Delegation zur Reduktion von Event‑Listenern, persistenter Warenkorb, stärkere Validierung der Eingaben und internationale Formatierung der Preise mittels Intl.NumberFormat.

### A10. Benutzer‑ und Entwicklerhandbuch (Auszug)

Abschließend enthält der Anhang einen kompakten Überblick aus der Benutzerdokumentation und dem README für Entwickler.

**Benutzerdokumentation:**

* **Starten des Shops:** Archiv Schmuck\_Webshop\_3\_0\_\_mit\_Produktdetails.zip entpacken und im Ordner Schmuck Webshop 3.0 einen lokalen Webserver starten, z. B. per python3 -m http.server 8000. Den Shop unter http://localhost:8000/index.html im Browser öffnen.
* **Navigation:** Im Katalog nach Produkten suchen, filtern und sortieren. Details werden in einem Modal angezeigt; Varianten können ausgewählt und in den Warenkorb gelegt werden. Der Warenkorb zeigt Netto‑ und MwSt‑Summen und ermöglicht einen Dummy‑Checkout.
* **Registrierung:** Über die Navigationsleiste gelangt man zur Registrierungsseite. Erforderlich sind Vor‑ und Nachname, E‑Mail‑Adresse und ein sicheres Passwort (mindestens 8 Zeichen, Groß‑/Kleinbuchstaben, Zahl und Sonderzeichen). Die Daten werden im Browser gespeichert und dienen nur Demonstrationszwecken.
* **Bedienung auf Mobilgeräten:** Die Oberfläche passt sich automatisch an kleinere Bildschirme an; bei Drehung des Geräts empfiehlt sich ein erneuter Seitenaufruf für korrektes Layout.
* **Datenschutz:** Sämtliche Daten werden ausschließlich im Browser gespeichert (Local‑Storage). Es findet keine Übertragung an einen Server statt. Zum Löschen der Daten kann der Local‑Storage im Browser geleert werden.

**Entwicklerdokumentation (README‑Auszug):**

* **Projektstruktur:** HTML‑Seiten (index.html, shop.html, checkout.html, register.html), Styles (style.css, theme.css), JavaScript‑Module (ui.js, products.js, app.js, auth.js) und die Produktdaten (products.json). Bilder befinden sich im Unterordner images/.
* **Entwicklung:** Der Code verwendet modernstes Vanilla‑JavaScript und Bootstrap 5.3. Daten werden via fetch() geladen, Event‑Delegation verbessert die Performance. Änderungen am Produktkatalog erfolgen direkt in der JSON‑Datei. Registrierung und Login sind rudimentär implementiert; Passwörter werden mittels Web‑Crypto‑API gehasht.
* **Erweiterungen:** Für eine produktive Version empfiehlt sich die Einführung eines Backends (z. B. Flask, Django oder Express) zur Verwaltung von Nutzerkonten und Bestellungen, die Integration eines Zahlungsdienstleisters sowie automatisierte Tests.

Diese zusätzlichen Anhänge stellen sicher, dass keine wichtigen Informationen in separaten Dateien verbleiben und erleichtern die Nutzung, Pflege und Weiterentwicklung des Schmuck‑Webshops.